AFRL-RV-PS-
TR-2012-0085 AFRL-RV-PS-
TR-2012-0085

RECONFIGURABLE PARALLEL COMPUTER ARCHITECTURES FOR SPACE APPLICATIONS

Marios S. Pattichis

The University of New Mexico
Dept of ECE
MSC01 1100
1 University of New Mexico
ECE Bldg., Room 125
Albuquerque, NM 87131-0001

7 Aug 2012

Final Report

AIR FORCE RESEARCH LABORATORY
Space Vehicles Directorate
3550 Aberdeen Ave SE
AIR FORCE MATERIEL COMMAND
KIRTLAND AIR FORCE BASE, NM 87117-5776

DTIC COPY
NOTICE AND SIGNATURE PAGE

AFRL-RV-PS-TR-2012-0085 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT

//SIGNED//
KEITH AVERY
Program Manager

//SIGNED//
KEN HUNT
Tech Advisor, Space Electronics Protection Branch

//SIGNED//
KEN D. BOLE, Lt Col, USAF
Deputy Chief, Spacecraft Technology Division
Space Vehicles Directorate

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YY)* 07-08-2012	**2. REPORT TYPE** Final Report	**3. DATES COVERED** *(From - To)* 1 Feb 2005 – 30 May 2012
4. TITLE AND SUBTITLE Reconfigurable Parallel Computer Architectures for Space Applications		**5a. CONTRACT NUMBER** FA9453-06-C-0211
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER 63401F
6. AUTHOR(S) Marios S. Pattichis		**5d. PROJECT NUMBER** 2181
		5e. TASK NUMBER PPM00004415
		5f. WORK UNIT NUMBER EF003210
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) The University of New Mexico, Dept of ECE, MSC01 1100, 1 University of New Mexico, ECE Bldg., Room 125, Albuquerque, NM 87131-0001		**8. PERFORMING ORGANIZATION REPORT NUMBER** TR-2011-0214
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory, Space Vehicles Directorate, 3550 Aberdeen Ave., SE, Kirtland AFB, NM 87117-5776		**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RVSE
		11. SPONSOR/MONITOR'S REPORT NUMBER(S) AFRL-RV-PS-TR-2012-0085

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for Public Release; distribution is unlimited. (377ABW-2012-1124 dtd 22 Aug 2012)

13. SUPPLEMENTARY NOTES

14. ABSTRACT
In this report, we are providing a summary of the two main efforts: (i) the work done on the AFRL-UNM High-End Reconfigurable Computer (HERC), and (ii) the Adaptive Wiring Panel (AWP). The AFRL-UNM High-End Reconfigurable Computer (HERC) is a new multiprocessor architecture for use in high-performance on-board space applications. This architecture was developed to have modular and compact basic processing nodes, interconnected by high-speed communications links that aggressively embed most of their components in Field Programmable Gate Arrays (FPGA). We also present an approach for developing the concept of a manifold of adaptive wiring cells connected as a single overall Adaptive Wiring Panel (AWP). The main use of the AWP is related to affordable plug-and-play space applications but the concept can be used for different applications. A reconfigurable switch fabric enables dynamic routing of signals and power for space systems.

15. SUBJECT TERMS
High-End Reconfigurable Computing, Adaptive Wiring Panel

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Keith Avery
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	Unlimited	108	**19b. TELEPHONE NUMBER** *(include area code)*

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. 239.18

(This page intentionally left blank)

TABLE OF CONTENTS

LIST OF FIGURES

Executive Summary

We are providing a summary of the two main efforts in this technical report. This includes the work done on the AFRL-UNM High-End Reconfigurable Computer (HERC) and the Adaptive Wiring Panel (AWP).

New requirements for on-board spacecraft processing systems demand solutions with high-end computation capabilities, reconfigurable logic for hardware acceleration and processing properties, as well as the ability to easily interface with different types of sensors and sub-systems. Additionally, it is desirable that the architecture of the systems be modular, easily deployable and versatile, while limited in weight and size.

This work describes the AFRL-UNM High-End Reconfigurable Computer (HERC), a new multiprocessor architecture for use in high-performance on-board space applications. This architecture was developed to have modular and compact basic processing nodes, interconnected by high-speed communications links that aggressively embed most of their components in Field Programmable Gate Arrays (FPGA).

We also present an approach for developing the concept of a manifold of adaptive wiring cells connected as a single overall Adaptive Wiring Panel (AWP). The main use of the AWP is related to affordable plug-and-play space applications but the concept can be used for different applications. A reconfigurable switch fabric enables dynamic routing of signals and power (power, digital, and analog signals can be routed for space systems. This concept can also be applied to terrestrial applications such as aircraft wiring and ground-based systems, for example dynamic routing of media such as light or fluids is also possible using the same fundamental switch architecture.

The AWP is a manifold of adaptive wiring cells cast as a single overall panel. The panel is a pegboard-like structure, which does not articulate specific sockets, but rather provides a continuous grid of contact pads and mechanical mounting holes. Implementation is based on three basic elements: (i) cell units (CU), (ii) a cell management unit (CMU), and (iii) modules. CUs are the minimum independent units of the AWP, each with interconnections and links with other cells to form the switch fabric by which we wire components to each other. The CMU talks independently with all CUs and manages the wiring path and panel switch connections. Inter-Integrated Circuit (I^2C) is the protocol used for all the communications. Finally, modules are the "widgets" that make up components to be wired (e.g. power supplies, gyros, thermistors, resistors, LEDs, etc.). The modules can be plugged in at any orientation, which is detected by the CUs. We present the results related to the current compact version of the AWP based on 5x5cm cell units. Some of the advantages of this version are the elimination of internal cables and the inclusion of I^2C repeaters.

1.0. Introduction.

Our final technical report is primarily focused on the description of our efforts in developing the Adaptive Wiring Panel (AWP) and the AFRL-UNM High-End Reconfigurable Computer (HERC). When describing the HERC, we refer to the Ph.D. dissertation by Jorge Parra [1]. Furthermore, much of the material for describing the AWP has been taken from [2][3].

1.1. AFRL-UNM HERC Prototype [1].

To investigate the properties of the HERC architecture, a prototype was constructed and studied. Benchmarking applications were run on this prototype to characterize the performance of its embedded microprocessors, its memory system, its communications network, and the performance of the system in a multiprocessor context. This analysis is accompanied by comparisons to related architectures that identify the position the HERC occupies in an architectural space. Also, analyses of the power and resources utilized by this architecture were performed.

This work defines a new multiprocessor architecture for use in high-performance, on-board space applications. It focuses on utilizing Field Programmable Gate Arrays (FPGA) and their embedded components as the primary devices for general-purpose multiprocessor computations. This work also develops the necessary hardware and software modules required to create the supporting communications network. Finally, it evaluates the resultant system in order to determine its strengths and weaknesses when used in this particular application.

The main innovations and contributions of this research are in the areas of multiprocessors, computer architecture, and space vehicles systems. They include:

- The definition of a new multiprocessor reconfigurable architecture for its use in space vehicles, utilizing FPGAs, and the modules embedded on them, as the main components of the system.
- A functional prototype of this architecture used to validate its feasibility, and to evaluate its final characteristics. Simultaneously, this functional prototype is a unique platform that can be used in the simulation and development of new complex designs, involving software and hardware for multiprocessor systems.
- The analysis of the performance of the main modules of the prototype, focusing on the relevant areas with respect to the new generation of space vehicles.
- Recommendations, based on the results found in the prototype, about the characteristics, the advantages and disadvantages that a reconfigurable multiprocessor, like the proposed one, should have in the context of hardware and software for space vehicles.

1.2. Adaptive Wiring Panel (AWP)

In vehicular platforms, the network of wires that connect sensors and actuators to other electrical boxes is referred to as a wiring assembly or wiring harness. The wiring harness plays a critical role in distributing signals and power throughout the platform. These wiring harnesses are intensively custom, often expensive, and can take a long time to build (i.e. months). As such, the wiring harness can be a limiting factor in the time necessary to build a new system from scratch, even if all components and software are available for the new design. Conventional spacecraft wiring harnesses are built with architectures that are fixed at manufacture. By implementing reversible (meaning they can be changed repetitively) and dynamically programmable software wires, we can form an “adaptive wiring manifold”. Adaptive wiring systems have many useful properties. They can, for example, be customized quickly, within seconds if the wiring configuration is known. They have tremendous flexibility in that they can be changed up to the last moment of a system’s development without removing components and performing painstaking rework. They also have the potential of self-healing and enhanced diagnostics through soft-definable probe signals.

Here, we explore the concept of an adaptive wiring manifold (AWM), a wiring harness that is reconfigurable and scalable for general applications. In principle, it is like a field programmable gate array (FPGA), which is a chip that can be programmed to implement wiring patterns, except that our adaptive harness allows for the routing of continuously variable analog, power, and microwave signals (FPGAs, in general, can only manipulate Boolean signals). Since an AWM is pre-built and soft-configured as needed, it supports the rapid development of platforms (such as spacecraft). We envision that this might be achieved by assembling a number of tile-like panels, each a “smart substrate” containing a portion of an AWM, to form the overall wiring harness of an entire platform. Components can then be mechanically and electrically attached to panels in the simplest cases completing a platform build cycle. In this manner, we can reduce the time from building some custom systems from months to minutes.

We will present the concepts for adaptive wiring architectures. Then, the architecture for cellular adaptive wiring manifolds. Next, we present the hardware and software details for the implementation of the current prototype built. We finish with the conclusions. We will use the words “manifold”, “harness”, and “assembly” interchangeably, though the latter term will also be used to refer otherwise to aggregations of components. The meanings should be clear by context in discussion.

2.0. Methods.

A synopsis is provided. For the complete details, please refer to [1].

2.1. AFRL-UNM HERC Prototype.

A digital system designed to be a high-end general-purpose processing module to be used on a space vehicle must comply with some special requirements. We have identified that these requirements could be satisfied with a new architecture based on High-End Reconfigurable Systems (HERCs). This architecture, which borrows characteristics from Massive Parallel Processors (MPPs), is made up of a collection of reconfigurable processing nodes interconnected by a high-speed communications network.

We introduce the AFRL-UNM High-End Reconfigurable System. We also elaborate on how this system satisfies the special requirements imposed by this particular application, and what portions of this architecture will be implemented in a prototype. Creation of this prototype has allowed validation of the feasibility of the proposed system, as well as an opportunity to study the advantages and disadvantages of its modules, in the context of the aforementioned application.

2.1.1. Definition and description of the AFRL-UNM HERC

The AFRL-UNM HERC is a reconfigurable, multiprocessor, general-purpose high-end computer. Its design is mainly focused on modularity and scalability while providing a reconfigurable, microprocessor-based platform. It was also conceived to explore the tradeoffs involved in designing a multiprocessor system constrained by the fact that all the main processing blocks are embedded in FPGAs.

Reconfigurability in this system is defined as its capability of partially changing portions of its hardware, upon user's request. This capability is inherited from the FPGAs on which this system is based, and is accomplished by changing the data used to program the interconnections between the resources available on the FPGA.

The main component of the system is the Basic Module (BM). It is a completely functional computational unit, composed of a Xilinx Virtex-4 FX100 FPGA, two DDR2 memory modules, a USB 2 host interface, Ethernet, System ACE, Analog-to-Digital Converters (ADCs), and four networks. A single BM can be used as the processing unit of a single-powered system or as the building block of a system with a medium or a large number of processing units.

2.1.2. The Interconnection Network

The AFRL-UNM HERC has four communication mechanisms implemented on each BM. The first mechanism is based on the Ethernet 10/100 protocol. Although this network is not initially intended to be the main communications support for HERCs with a large number of BMs, analysis of its performance was studied. The Ethernet network is also intended for communications with external networks, to broadcast commands to every component of the system, or to be the main communications port in systems powered by a single BM.

The Rocket I/O-based connections form the main communication medium of the ARFL-UNM HERC. It is a distributed, message-passing network fully linking a BM with its neighbors in a mesh structure. The advantages of such networks are bandwidth, scalability, and fault tolerance.

As for the second mechanism, a BM interconnects a group of BMs on other levels as well as those on its own level. This structure also offers advantages when hierarchical configurations or data distribution configurations are implemented.

The other two communication mechanism implemented on the AFRL-UNM HERC are a common bus structure and the JTAG connection. The common bus structure is a 24-bit bus implemented in clusters of BMs on the system. Its function is to carry low-speed commands and broadcast messages. Finally, the JTAG network serially connects all System ACE and FPGAs in the system.

2.1.3. AFRL-UNM HERC Microarchitecture

The microarchitecture of the AFRL-UNM HERC refers to the hardware portion that is necessary for the system's proper operation and that is embedded in the Virtex-4 FPGA fabric. It is divided in three main portions: Peripheral interfaces, inter-board, and inter-core communications. The Peripheral interface groups the circuits intended to interconnect the processor and the peripherals.

Each Basic Module features two PowerPC microprocessors, and two DDR2 memory controllers attached to the Peripheral Local Bus (PLB). These two elements form a minimal functional core linked to the rest of the system through the inter-core connection, the inter-board communication links, the common network, and the JTAG chain.

It is important to mention that the microarchitecture design is dependent on the FPGA performance and the availability of its resources. This design has to account for the reasonable use of these resources, saving enough of them for the implementations of the applications used in the HERC's mission.

2.1.4. Operating System and Supporting Software

The AFRL-UNM HERC is a message passing system hosted by a Linux kernel as shown in. Montavista Linux, a Real-Time Operating System (RTOS), was chosen as the operating system to run on each BM. The Linux kernel interfaces with the Rocket IO network, the JTAG network, and the Common Network through custom-designed drivers. To provide message-passing capabilities OPEN MPI, a Message Passing Interface platform, was chosen. It is based on Remote Shell (RSH) or Secure Shell (SSH) and a TCP/IP stack hosted by the operating system.

2.1.5. The AFRL-UNM HERC and the special constraints for an on-board system for space vehicles

The proposed HERC is a modular system, whose computing power scales as more Basic Modules are added to its network. Simultaneously, its network bandwidth scales when new BMs are added. This property simplifies the system's deployment and increases its versatility in diverse mission scenarios. Therefore, simple missions can use a HERC with few Basic Modules, while complex missions can require a system with a large number of them.

The reconfigurable properties of the proposed HERC, its multicore and its multiprocessor capabilities, qualify it to be able to process high-end software, enhanced with hardware acceleration. In this scenario, systems implemented in the HERC can exhibit independent hardware processing modules, hardware tailored coprocessors, dynamically reconfigurable systems, and other types of data processors (i.e. systolic systems). These distributed computational capabilities can be exploited to implement systems with error mitigation capabilities.

Hardware reconfiguration also leads to hardware reusability. As an example, a modular hardware system can be used in a first stage of the HERC's mission. When that stage finishes, some hardware modules can be reconfigured, and recycled in following stages of the mission. This characteristic is very attractive for space systems, where reduced hardware dimensions and weight are important constraints. However, it is important to mention that reconfiguration in these systems are achieved at slow rates.

The proposed HERC features a distributed, robust and flexible communications network. The multiple links that each BM features create redundant communication paths, useful in the event of BM failure and recovery. This network also allows configuring BMs that are only dedicated to data processing tasks, and others dedicated to peripheral interfacing. As is the case with the peripherals featured in the prototype of the Basic Module (the USB2, the System ACE, and UARTs), BMs featuring other interfaces and peripherals can be built, and linked to the network. Also, the FPGA IO capabilities can be used to interface to legacy systems, allowing peripherals that were not originally created for use with this system, to be hosted in the AFRL-UNM HERC.

The proposed HERC uses standard interfaces and commercially available tools. Its operating system is fully Linux compliant, allowing any software developed in any Linux distribution to be easily ported into this platform. The system utilizes standard GNU C and C++ compilers and libraries. As mentioned earlier, it uses OpenMPI as the high-level message-passing engine. This allows users to develop applications for the HERC in standard Linux multiprocessor architecture running OpenMPI. Similarly, the system's hardware portion was completely developed using commercial Xilinx, which allows hardware developments based on any Xilinx development board to be easily ported to the HERC.

2.2. AWP architecture.

2.2.1. Adaptive Wiring Architecture Concepts.

In the current wiring concepts with the exception of the plug-and-play architecture, the harness configurations are fixed and cannot be changed easily. In the case of the plug-and-play architecture, we reviewed a limited form of adaptiveness, in which a number of modules could be freely commuted on panels, and the panels themselves can be composed to form a larger system. In this section, we present a far more powerful form of adaptive wiring architecture. In this architecture, we demonstrate a much greater flexibility in the types of modules, their termini count and arrangement, and the ability to reform the wiring to accommodate faults, testing, and repurposing to meet different needs.

2.2.2. Conceptual Architecture.

This subsection describes a number of basic principles for adaptive wiring systems. To introduce the basic idea, an abstract adaptive wiring structure (Fig. 1) that could be referred to as a panel or substrate contains a number of input/output (I/O) termini. These are shown in Figure 1(a) as connection points on the left (AI, BI, …) and right (EO, DO, …) edges. In the case of adaptive wiring, we can form connections between the termini "on demand". We can supply a wiring "problem" that we wish to solve, a set of termini that we wish to connect together. Through some (not shown) control mechanism, we can convey commands into the panel that cause it to form "virtual" wire connections as desired. In Figure 1(b), for example, we show the solution of virtual wires needed to connect termini together bearing the same pre-fix label input on the left edge to the corresponding label output on the right edge (e.g., "AI" connecting to "AO", etc.). In general, the wiring problem to be solved can be referred to as a netlist specification. In fixed wiring systems, netlists are implemented physically in the form of a wiring harness. In this example, the adaptive wiring system, represented as a "cloud" within the substrate, forms the wiring dynamically (under program control). At this point, the adaptive wiring concept is notional, and we have not suggested how the "cloud" is implemented.

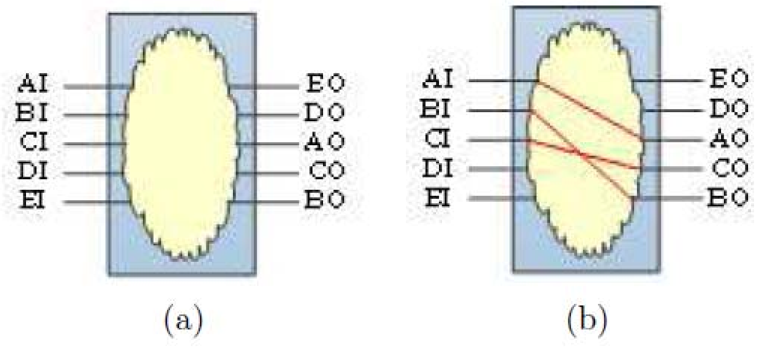


Figure 1. Basic concept of an adaptive wiring cell.

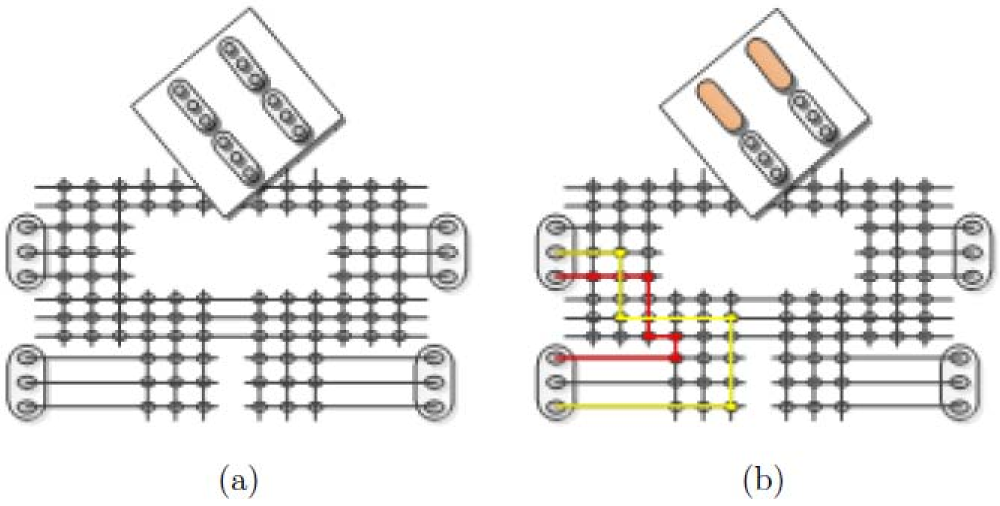


Figure 2. A physical wiring problem example.

Figure 2 depicts a notional implementation concept to provide some intuition about how an adaptive wiring system might actually be implemented. In this case, the substrate takes on the aspect of a physical panel featuring four sockets where components or “modules” can be mounted (Fig. 2a). To implement the amorphous “cloud” of wiring resources in Figure 1, a deliberate configuration is depicted consisting of a matrix of wires in rows and columns, with circles shown at the intersection points [4]. The circles represent electrical switches that, when closed, short together the associated row and column. We are not immediately concerned over the specific medium for switches. They could be, for example, metallic relays, solid-state switches, microelectromechanical systems (MEMS) devices, or combinations of these and other switch types [5], [6]. Using such a fabric, implementing a solution to a particular wiring netlist amounts to closing a number of switches, as shown in Figure 2(b), which shows how a netlist problem having two “virtual wires” or nets (involving connections between two placed modules) might be solved (through a total of eight switch closures).

We can extend the concept through an approach analogous to the segmenting previously described. A number of substrates could be tiled together, connected through some of the available external termini as suggested in Figure 3. In this case, two adaptive wiring substrates or “cells” form an extended system. Now, a netlist solution is compound in nature, involving a global specification (such as “connect AI to AO”) and local specifications (the specific solutions of each “cloud”). The local specification involves allocating terminals between cells, and then defining subnet problems for each cell. It is then necessary to compute local solutions within each cell to implement the implied sub-netlist. It is obvious upon inspection that there are many non-unique solutions for a particular global netlist problem, both in terms of the allocation of nets between cells and implementations of the sub-netlists within each cell.

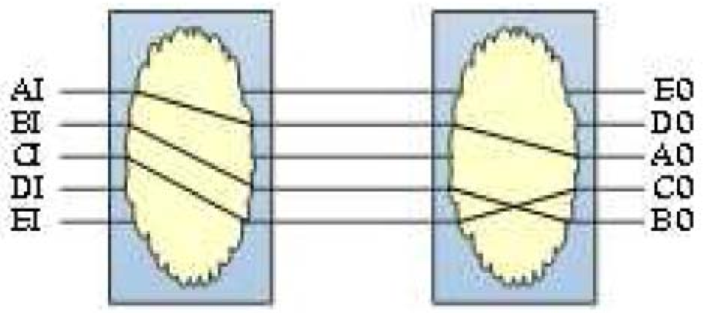


Figure 3. Extended adaptive wiring system by using two cells.

2.2.3. Benefits of Adaptive Wiring.

As previously discussed, adaptive wiring manifolds offer a number of benefits in developing new systems. Since the adaptive wiring substrates (or panels) may be pre-built and inventoried until use, it is possible to retrieve them as needed and configure them on demand. Rather than wait for custom-defined wiring harnesses to be developed and delivered, a process that could take weeks or months, the adaptive versions can be configured very quickly. Unlike custom wiring harnesses, whose wiring pattern is permanently locked in, adaptive panels can be altered as needed to accommodate late-point changes.

Adaptive wiring systems furthermore, have two powerful benefits that are impossible in any other wiring technology. First, this architecture has the ability to adapt to faults that occur after a system is placed in the field. Since wiring patterns can be software-definable, defects could conceivably be fixed by computing an alternate configuration. In Figure 4, a faulty connection between B1-BO can be rectified by configuring other wiring resources that can achieve an equivalent connection without removing a system from the field (which is often impractical, as is the case for space systems).

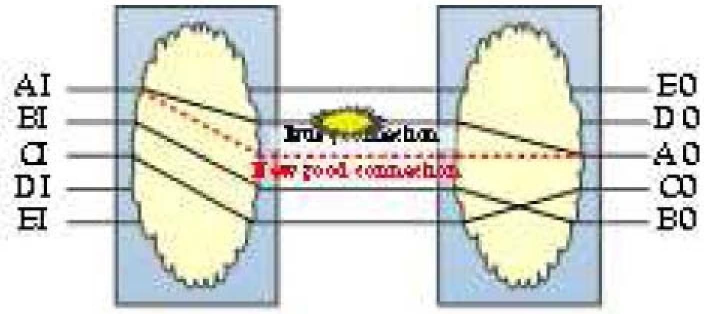


Figure 4. Example of the ability to adapt to faults.

The second unique advantage of adaptive wiring systems is the ability to form probe connections for diagnostic and maintenance purposes. Temporary probes can be inserted at normally inaccessible buried nodes within a wiring system and removed from the system software to be no longer used. This concept is depicted in Figure 5. In this case, we use the adaptive wiring system to set up a temporary connection to check a possible problem with terminal CO on the right panel.

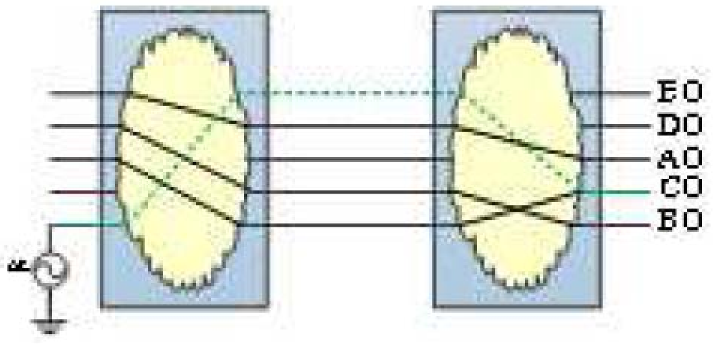


Figure 5. Example of manual diagnosis of connections

The techniques demonstrated in Figures 4 and 5 can be combined with algorithms to form a self-healing system. Self-healing, as an "active" concept, can be viewed as having two phases, the first being diagnostic, the second being restorative. Clearly, the use of temporary probes can serve to probe an adaptive wiring system, even in situ, to explore the continuity of wiring resources. Upon discovery of defects, an algorithm in the real-time system can compute an alternative wiring path. In earlier AFRL-sponsored work [7] we learned it was possible to achieve self-healing as a linear-time process in an active system.

2.2.4. Architecture Description of Cellular Adaptive Wiring Manifold.

In this section, we make the concepts discussed more concrete through an example of a cellular adaptive wiring manifold. It extends the notions of the scalable adaptive wiring harness, drawing inspiration from the panelized construction of the PnPSat-1 platform.

2.2.5. Cellular Adaptive Wiring Architecture Overview.

A simplified physical depiction of an adaptive wiring panel is shown in Figure 6. The Adaptive Wiring Panel (AWP) is a panel, a pegboard-like structure, which (unlike Fig. 2 or 1) does not articulate specific sockets, but rather provides a continuous grid of contact pads and mechanical mounting holes. It is a planar substrate composed by tiling together a number of cells. Each cell is conceptually similar to the abstracted cells shown in the two-cell concept in Figure 3, tileable to form arbitrary panels, such as the 8×8 cell panel shown. Each cell contains termini, both around the edges (for inter-cell connections) and on the top surface. These latter termini (pins) are the only ones that most users would experience in creating a system.

To "use" the panel, a number of modules can be arranged on the panel surface and attached. These modules connect to a number of the panel pins, and in this sense, the bottom surface of modules can be thought of as their electrical connector. As such, this architecture implements the notion of a surface-mounted, blind mated connector. However, this approach allows tremendous flexibility over a traditional blind-mated connector in that modules can be placed in many locations and any of the four "Manhattan" directions (orientations).

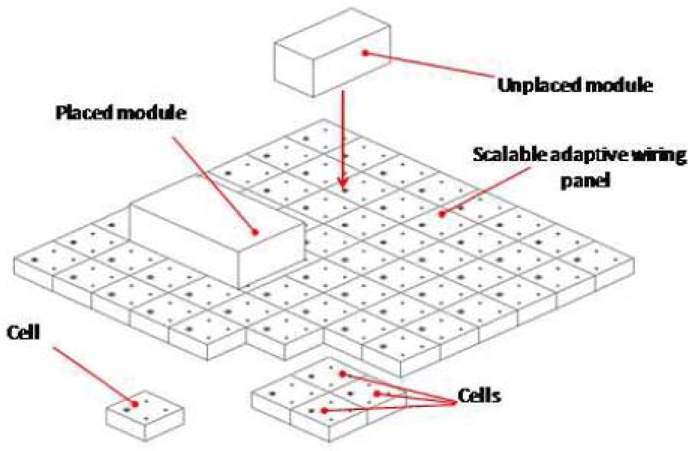


Figure 6. Cellular implementation of adaptive wiring panel.

Once the modules are placed as desired, the netlist specification for their interconnections is fed to computer referred to as the cell management unit (not shown in Fig. 6), which computes the configuration for the virtual wiring in the array of cells that implement the desired wiring solution.

Therefore, the AWP implementation is based on three basic elements: (i) cell units, (ii) a cell management unit, and (iii) modules. Cell units are defined as the minimum independent unit of the AWP, all with interconnections and communications with other cells, forming (by iterative tiling) the switch fabric by which we wire modules to each other. The Cell Management Unit communicates independently with all cells and manages the wiring path and switch connections of the panel. Finally, the modules, being the "widgets" that make up components to be wired, contain some features to make it possible to integrate them efficiently (such as a small processor).

2.2.6. Cellular Wiring Grid Convention.

In the adaptive wiring panel, it is essential to identify conventions relating to mounting and electrical distribution, such as those shown in Figure 7. We define the adaptive panel as a series of intercalated grids, namely "mechanical", "power", and "signal". Of these, the mechanical grid is the coarsest grid. The mechanical grid defines the attachment locations for physically mounting modules. We show the points on this grid as occurring at a density equivalent to the pitch of a "unit cell" (in our definition, we specify this to be 5 cm, although there is nothing special in this choice). The power grid is defined by superimposing a 2.5cm grid onto the mechanical grid (by convention, the point belonging to the coarsest grid "wins", and the coincident points of finer superimposed grids are suppressed). The intended purpose of these grid points is the support of higher current wiring, consistent with those associated with the delivery of power. The signal grid is defined at four times the density of the mechanical grid (1.25cm pitch), which is intended to be the most common case in general-purpose wiring.

For cellular implementations of the adaptive wiring panel, it is convenient to render cells as integer multiples of the unit cell dimension. The physical tile boundaries are offset (as shown) to avoid cutting through grid points.

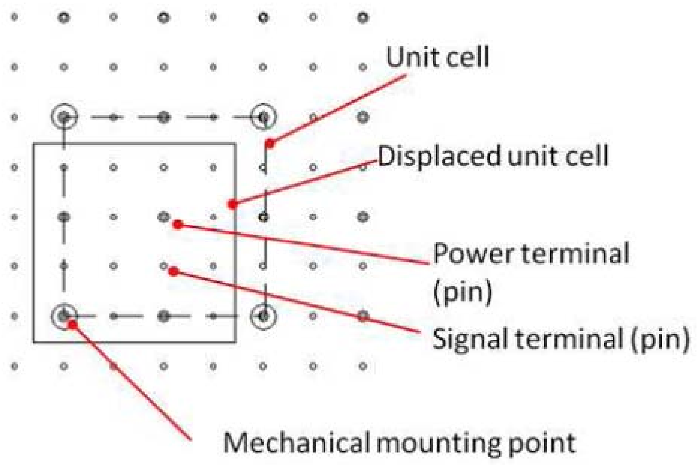


Figure 7. Conventions within adaptive wiring panel.

3.0. Results and Discussion.

Here, we discuss results on the AFRL-UNM HERC Prototype and the adaptive wiring panel (AWP).

3.1. AFRL-UNM HERC Prototype.

The AFRL-UNM HERC Prototype was completely built and it is described in Appendix C. For complete details, please refer to [1].

Not every detail of the AFRL-UNM HERC definition was implemented in the first prototype of this system. Some of its portions were left for future development and research. The main objective of the development of this prototype was to obtain a hardware and software platform that allowed the study of key characteristics of the architecture. Simultaneously, this prototype was used to validate the practical use of its components in a space vehicles context.

The first prototype is a HERC composed of two Basic Modules (BM). Although implemented in hardware, the USB2 module was not debugged and incorporated to the operating system. The System ACE, although completely developed and tested, was not used as the main media for the operating system. A faster solution, requiring less hardware, was found using a ramdisk to host the Operating System (OS), the filesystem, and using the JTAG port to load them.

The first prototype did not use all the Rocket IO links and the topology proposed for the Basic Module. A more simple connection, where every Rocket IO pair in a board was connected with another pair in a second board was implemented. This, given the restriction of the number of BMs built, provided the opportunity to study the behavior of the system when a large number of Rocket IO links were used.

The JTAG network and the common network were implemented in the first prototype's hardware, but were not linked to its OS. As mentioned before, the JTAG chain is being used to upload the bitstream containing the hardware hosted by the FPGA, its operating system and filesystem, and for debugging purposes. It is left for future versions of the system to implement the failure tolerance characteristics that the HERC should exhibit using the common bus system and the JTAG network.

Finally, the network interface module (also referred as network switch) was implemented as a mapped IO peripheral connected to a fast, general-purpose bus. Its functionality was partially embedded in hardware, and in a Linux networking driver. Ideally, this component should be designed as an intelligent coprocessor directly linked to the microprocessors through a dedicated high-speed port. Simultaneously, it should exhibit Direct Memory Access (DMA) characteristics including the necessary hardware to avoid data consistency problems. Xilinx FPGAs feature a Fast Simplex Link (FSL) port used to attach high-speed peripherals or coprocessors, and an Auxiliary Processor Unit (APU) Interface with an APU controller to assist this type of implementation. However, these interfaces are not completely supported in the Linux kernel, and its inclusion is a mayor development effort, out of the scope of this research.

Appendix C describes more in detail the first prototype, its development and the details of its construction. It is useful for those readers interested in replicating the system, or building a new one following the recommendations included in this manuscript.

3.2. Adaptive Wiring Panel.

3.2.1. First Example of Adaptive Panel Design and Implementation.

In this section, we will describe the implementation of a demonstration adaptive wiring panel (AWP) system shown in Figure 9. The demonstration system (right) contains six (of 64 planned) cell units, a few simple modules (for plugging into the adaptive panel), and a laptop as the controlling cell management unit (CMU). As a programmable fabric, the AWP requires tools to generate specifications for implementing particular wiring solutions to interconnect "modules" (shown lower left). A simple graphical user interface (GUI), shown upper left, was created for this purpose.

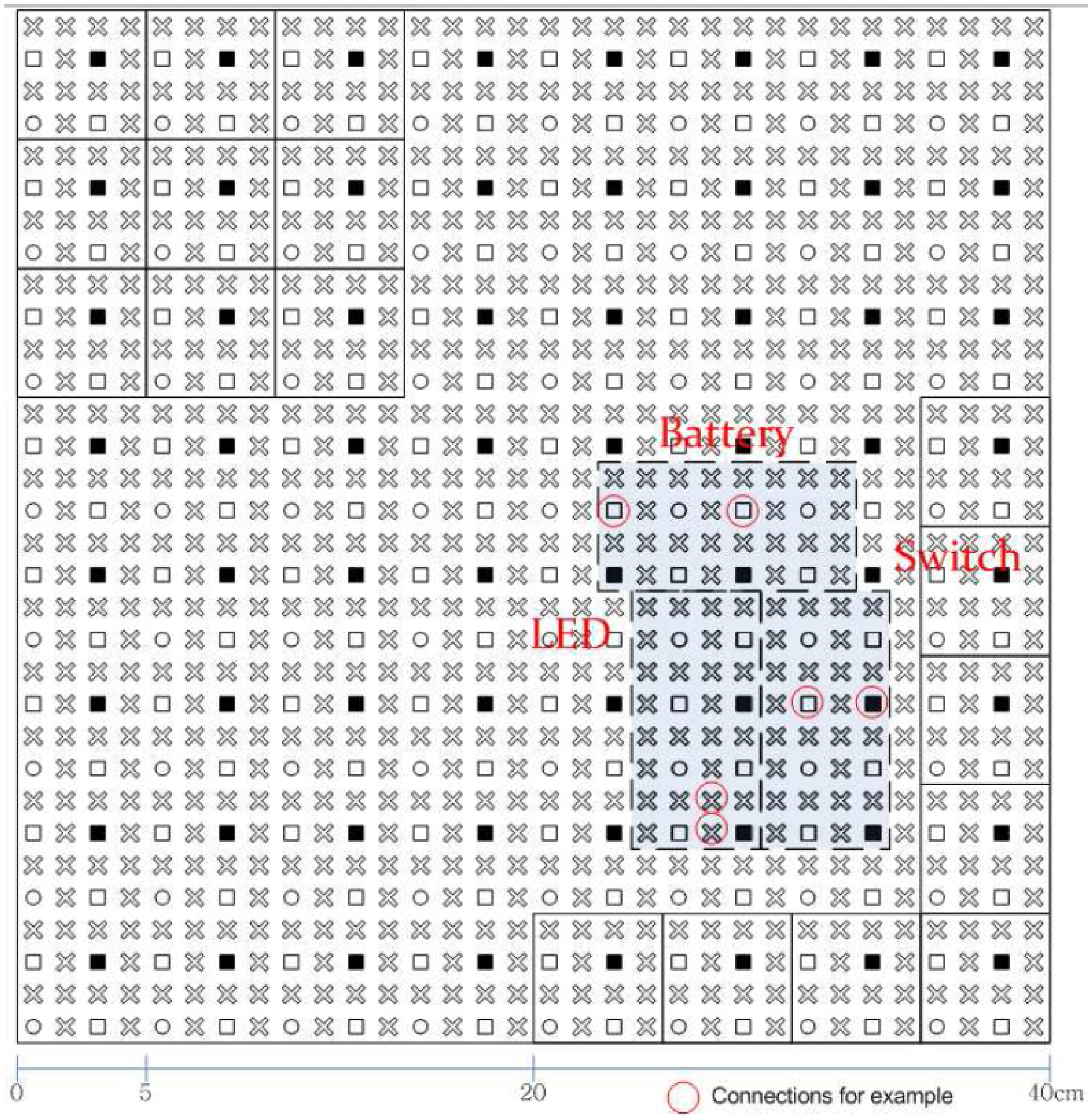


Figure 8. Adaptive Wiring Panel (AWP) concept using 64 cells in a 8x8 array

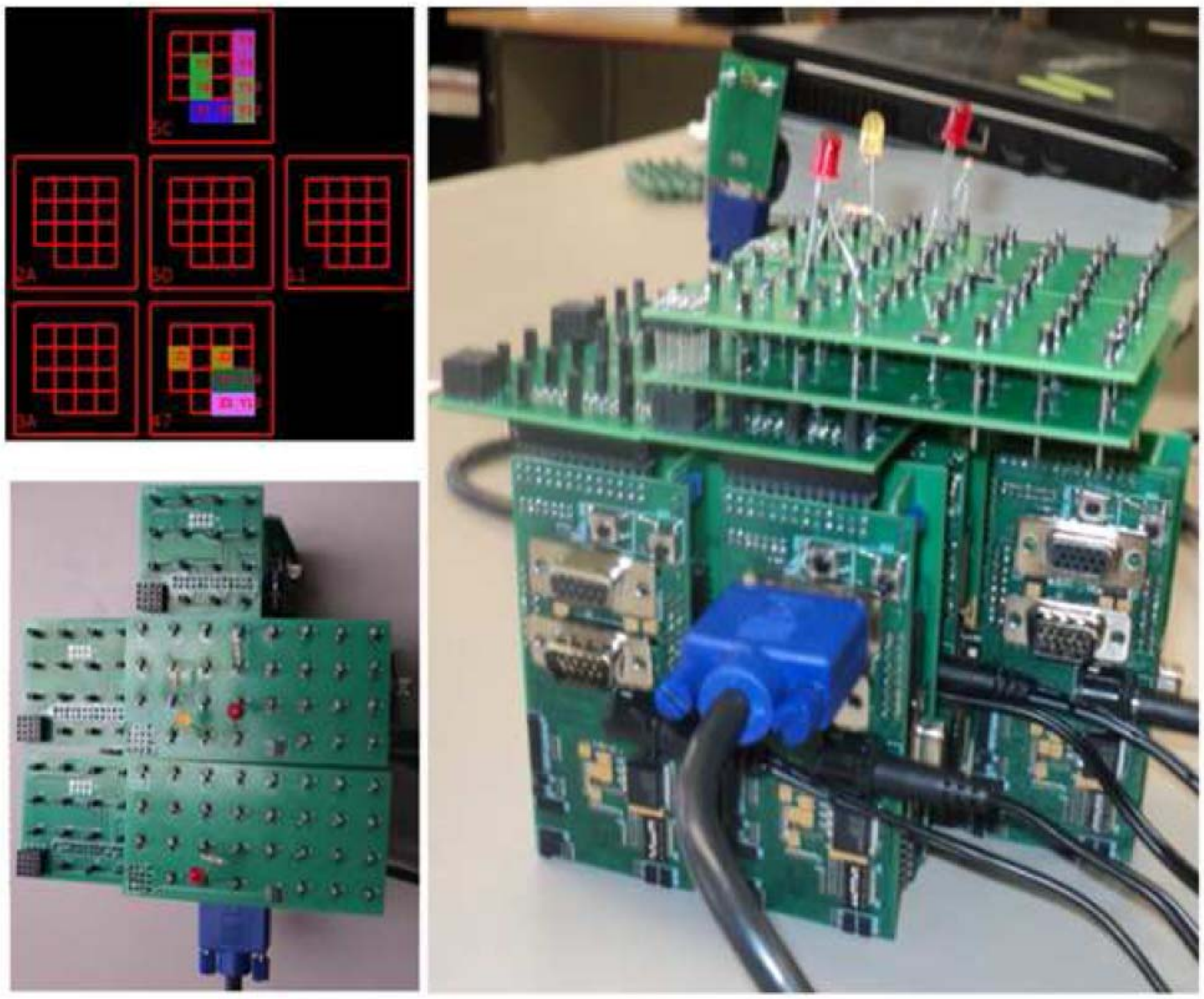


Figure 9. Partial adaptive wiring panel (AWP) system, containing 6 cells and 2 modules.

The main objective is to design, develop and build a 40×40cm adaptive wiring panel (AWP) and the modules to be connected in there. For a first stage, both signal and power connections are programmable. The mechanical connections are fixed. In Figure 8, we show the proposed 40×40cm panel with the connectors. The panel shown is square-shape with three types of connectors on it: (i) signal connectors (for transmission and reception of the signals, ‘X’ connectors in Figure 8), (ii) power connectors (for connecting to the source powers, ‘ ’ connectors in Fig. 8) and (iii) mechanical (mech) connectors (for holding the modules, ‘ ’ connectors in Fig. 8). The power connectors with black color ‘■’ represents fixed connections to ground (GND).

Figure 8 shows an example for connecting three modules on the adaptive wiring panel: (i) a module with a LED, (ii) a module with a switch (to control the LED from (i)) and (iii) a battery module. The modules are connected on the surface of the panel. They can be placed at any location within the panel. Each module needs to be connected in the locations shown with a red circle. We will next describe the cell unit, the modules, and the CMU that manages the configuration of the overall system.

3.2.2. The AWP Cell Unit.

Each cell unit (CU) is an “atomic” element, a minimum independent unit required to create the AWP. Its logical architecture is shown in Figure 10. A rectilinear tile that connects to its nearest neighbors in all four directions is referred to as a “NEWS” (north-east-west-south) network. Each edge (detailed only for the “east” port) contains a local communications port (for inter-tile information sharing), as well as pins for routing wiring connections between other edges and the primary surface array of contact pins. The surface array is the primary set of termini that are user-accessible. These are intended to support connections to matching pins present on “modules”, which are to be surface mounted onto an AWP. “Modules”, as will be discussed, are intelligent assemblies, and as such, require communications. The cell-module I^2C port provides support for this purpose. Finally, a single “cell common I^2C port” is provided to support communications to the cell management unit. Unlike the other I^2C ports, which are implemented as point-to-point interfaces, the common I^2C port is connected to all cells in an AWP.

The cell functions are managed by a “cell local processing unit” (fully implemented in hardware using FPGAs), including the six communications ports, cell status functions (such as maintaining a globally unique identification code), and configuring the switches connecting the wiring resources in the cell. (additional details are described in [2]). The functions of each CU are:

1. Control the programmable connections of the AWP.
2. Communication with (up to) four neighbors: each CU needs to communicate with its physical neighbors to recognize spatial orientations.
3. Read “electronic datasheet” information from modules (each module has a probe pin which sends module specifications to the cell it is plugged into).
4. A low current power supply will be sent to power the modules to enable transmission of electronic data sheet information through predefined probe pins.
5. Communication with the cell management unit: each cell unit transmits and receives information to/from the cell management unit (i.e., cell units aside from neighbor recognition cannot communicate directly with each other). Each cell block, upon system power up, will send identification information such as ID of the CU, the IDs of its neighbors and relative orientations, and module Electronic Data Sheets, if connected to that CU.

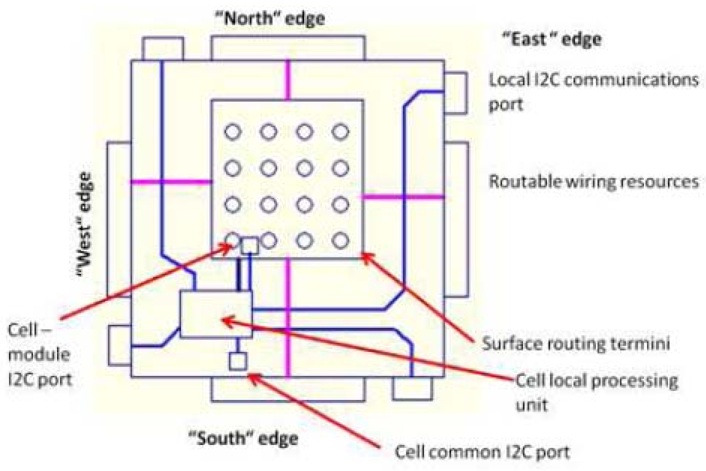


Figure 10. Cell unit logical architecture

Figure 11 illustrates the physical embodiment of an AWP cell unit in our prototype. Each AWP cell consists of 5 boards:

1. Top board is where the modules are placed.
2. South board is the main board where the main hardware is placed: a FPGA with all the logic control, the relays to close the connections, and extra hardware (for example, to reconfigure the FPGA for updates of the system). This board controls the rest of the boards. It includes a connector that connects to the North board of a neighboring AWP cell.
3. East board is connected to the West board of a neighboring AWP cell.
4. North board is connected to the South side of a neighboring AWP cell.
5. West board is connected to the East side of a neighboring AWP cell.

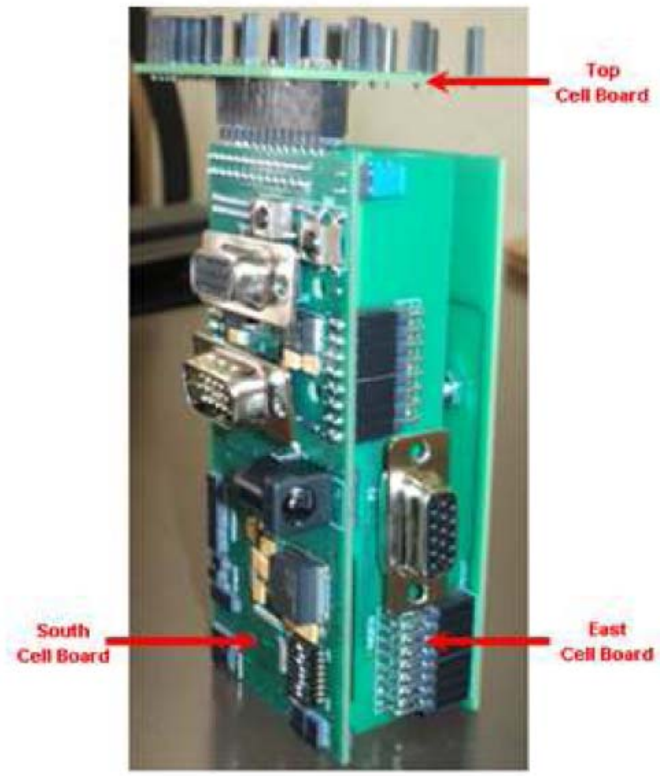


Figure 11. Cell unit of the AWP

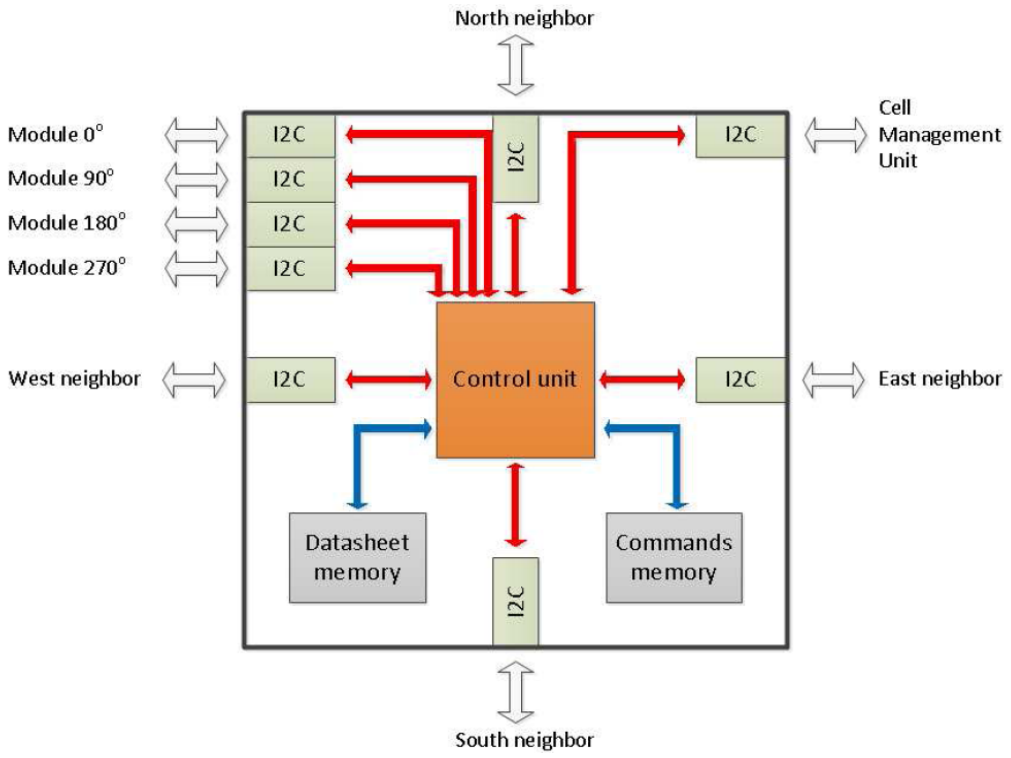


Figure 12. Block diagram of the hardware design in each FPGA cell unit.

The CU has been fully implemented in a FPGA using VHDL. The CU hardware design is depicted in Figure 12. It consists of a main control unit, internal memory units, and I^2C blocks to communicate with every other component of the AWP.

The I^2C blocks required (all of them designed and implemented in the FPGA) are: (i) 4 for communications with the neighbors, (ii) 1 for communications with the CMU, and (iii) for communications with the module. The objective of the last 4 I^2C blocks is to detect not only the module but also its orientation: 0°, 90°, 180° or 270°.

The CU unit can be divided in three parts:

1. Cell block board: it consists of:
   - Cell block pins (Y1-Y12, Z1-Z3).
   - I^2C connector: It is utilized to send information from the module to the cell. An I2C connection is used.
   - Mechanical connector: Since we need to detect the module orientation, we use an array of four I^2C connections (16 pins). Only one connection is active though. When the cell block is rotated, the 'rotation board' 2×8 connector pins (that remain fixed) will have a different pin configuration that will enable the FPGA board to detect a different module orientation (see Figure 13).
2. Rotation board: the mechanical connector has 4 × 4 pins so that it is possible for the FPGA board to detect the module orientation (see Fig. 23). This 'rotation board' translates the cell block rotations into a set of 4 I^2C connections so that one specific rotation activates a specific I^2C connection.
3. Relays board: it provides the following functionality:
   - It houses the 71 relays that are controlled by the FPGA.
   - It routes the cell block pins (Y1-Y12, Z1-Z3).
   - It transfers information (signals and power) between different neighbors. These connections are for the ports X1-X16, U1-U3, and GND.

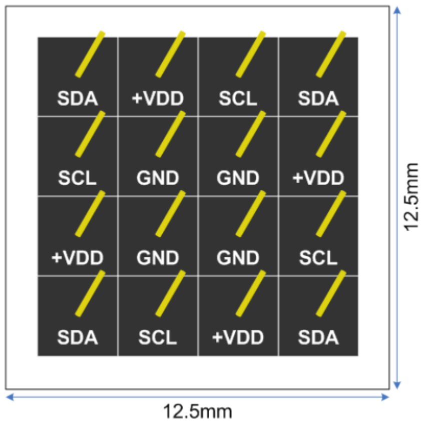


Figure 13. Mechanical connector with 4 sets of I^2C connectors to detect the modules.

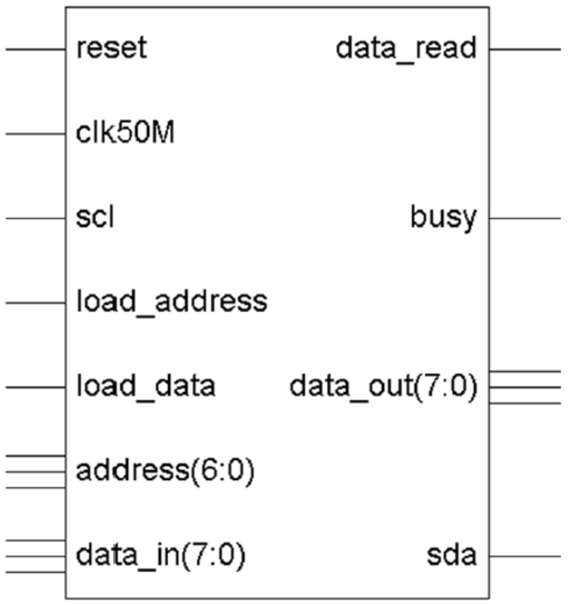


Figure 14. I^{2}C block to be placed in the cell blocks

In terms of the I^{2}C block, Figure 14 shows the block with inputs and outputs, for the I^{2}C block designed. The inputs are shown on the left and the outputs on the right (except for SDA that is input-output signal). We describe each signal in Table 1. This block is based on a state machine to control the seven states in an I^{2}C communication:

1. idle: no communication has been started,
2. send address: the CMU sends the address of the cell block to be read,
3. read/write: the CMU sends the signal to define a reading or a writing operation,
4. acknowledge: if the address sent by the CMU is the same as the cell block, the cell block places a low signal in the SDA line,
5. send data: if the acknowledge was true, the data is transmitted during this time. When the read command was sent by the CMU, the cell block sends the information using the SDA line; otherwise (write command) the CMU sends the information using the same line,
6. acknowledge: if an acknowledge signal is sent again, a new data is sent (going back to the state (v)), and
7. stop: when the communication is finished, the CMU sends a stop signal using both the SDA and the SCL lines.

Table 1. Description of the ports of the I^{2}C block designed using VHDL

Port Name	Direction	Size (in bits)	Function
reset	Input	1	Reset the system. This asynchronous input sends all the internal and external signals back to the default values.
clk50M	Input	1	Clock signal. The Spartan3E board uses a 50 MHz oscillator. The input frequency is internally reduced to 100 KHz to work with the standard I^{2}C adopted.
address	Input	7	Address to be set up in the cell block. We are using a 7-bit address size.
load_address	Input	1	Load the address. When this input is high, the address defined using the 'address' input is set up into the cell block.
data_in	Input	8	Data to be transmitted from the cell block to the CMU.
load_data	Input	1	Load the data. When this input is high, the data defined using the data_in input is stored in the cell block.
SCL	Input	1	Serial clock. This signal is controlled by the CMU. We are using a signal of 100 KHz.
SDA	Input	1	Serial data. This signal sends the information between the CMU and the cell block.
data_out	Output	8	Data received. When the cell block has read the data sent by the CMU, the data is output using this port.
data_read	Output	1	Data received ready. When the cell block has finished reading the data from the CMU, this output is high.
busy	Output	1	This signal is asserted high when a communication is performed. On the other hand, when no communication is performed, this signal is low.

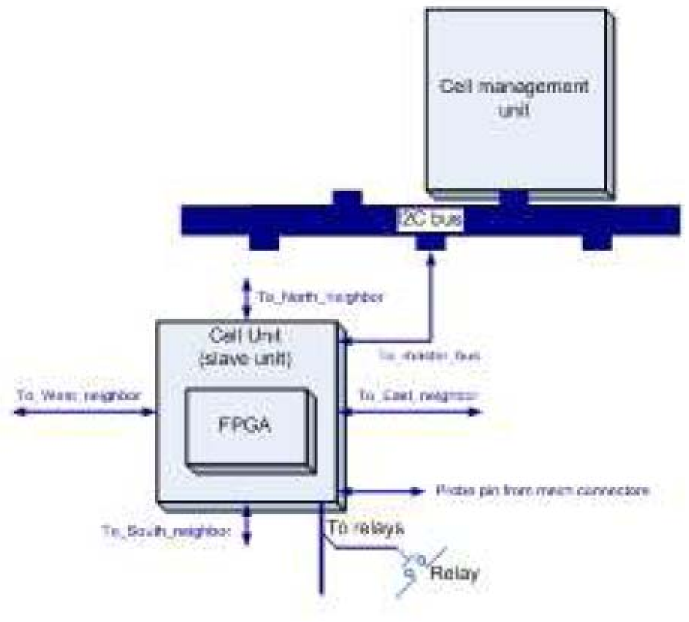


Figure 15. Cell Management Unit (CMU) with a typical connection to a Cell Unit

3.2.3. Cell Management Unit.

The cell management unit (CMU) manages global communications and routing configurations of all cell units on the AWP. An architectural diagram depicting the connection to a typical cell unit is shown in Figure 15, which indicates the chain between the cell management unit (global), the cell unit (local), and particular relays (switches) controlled by particular cells.

In the current demonstration system, the CMU is implemented as software running on a Linux machine. The basic setup for a GUI is shown in Figure 8 (top left) used to debug the mechanical connections and other design issues. The functions of the CMU are:

1. Manage the communication bus (I^2C protocol).
2. Read and organize initial location/orientation information from the CUs.
3. On a periodic basis, scan for any modules connected to a particular CU, and read embedded module "electronic data sheets".
4. Compute the global and local routing/path connections necessary to implement a desired netlist. The CMU has a prior knowledge of the layout of relays in each cell unit (they are identical in our demonstration system, but in principle the relay distribution can vary amongst cells). When the geometry of the AWP is formed (by assembling a tiled arrangement of cells), a complete graph related to the AWP and the connections is built by merging the subgraphs of each individual cell. After reading the module "data sheets" describing module contents and terminal information, the CMU attempts to compute connection paths specified between the corresponding terminals on the various modules plugged into the AWP.
5. Command individual cell units to open or close specific relays.

3.2.4. Module.

Modules refer to components that are plugged into AWP assemblies. Before we describe them, it is insightful to consider how an AWP might be used in a simple design example shown in Figure 16. A prospective AWP is shown with three modules in Figure 16(a) (a light bulb, a switch, and a battery) placed on the panel. The placed modules cover a number of pin locations and mechanical attachment points (revealed in Fig. 16(b)). These modules can be placed in any Manhattan direction and in any linear position so long as the mechanical attachment grids of the module align to those on the AWP. At this point, the AWP does not "know" what to do with these modules. Rather, the user placing the modules must supply this information in the form of a netlist. When this is done, the AWP can connect the modules by forming virtual wires, as shown in Figure 16(c). If a second copy of a module (e.g., an extra light bulb) is placed on the AWP, it intrinsically has the ability to connect to this second copy when the failure is detected (Fig. 16(d)).

We now move from this abstract description of an AWP with modules to describe our demonstration implementation. The modules we built were 5×10cm (or 2 cell units) with 24 signal connectors, 6 power connectors and 2 mechanical connectors (most of which need not be connected for a specific module type).

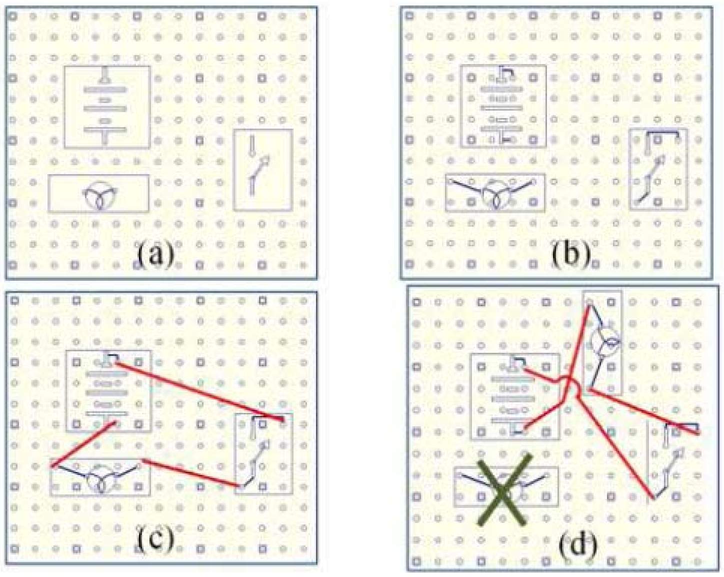


Figure 16. AWP in use.

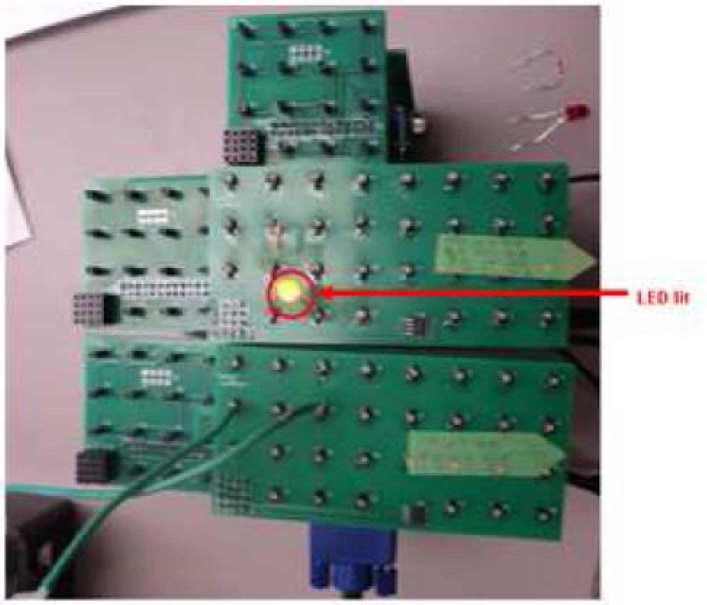


Figure 17. AWP with the circuit connected using two modules.

The modules employ an electronic datasheet on a small processor resident on the module (but not otherwise a part of the component(s) described). The datasheet of each module employs the SPICE language for the netlist descriptive format [8]. It is downloaded over one of the I^2C ports, interfacing to one of the cell processors contained in the panel, eventually being routed to the cell management unit. The CMU manages the database of modules and netlist connections, forming virtual wires on demand as needed (when it is possible to do so).

Figure 17 shows an example of two modules connected to the AWP: the bottom module with a battery and the top module with a resistor and a LED (acting as the light bulb in Fig. 16).

3.2.5. Routing Algorithm.

As described in [5] (for example), the wiring configurations of the AWP can be described as graphs, and manipulated with graph algorithms to find satisfying assignments for solving routing problems, as done routinely in FPGA synthesis algorithms. The basic heuristic algorithm used in the demonstration system can be summarized in the following pseudo-code:

```
Routing algorithm:
    Generate the graph based on the current cell configuration.
    Generate a Johnson-Trotter14 ordering of the required connections.
    Apply shortest-path algorithm for each connection.
    Update graph.
```

The algorithm begins with the graph representation of the current AWP configuration. Cells are sub-graphs of an overall extended (multi-cell) AWP. If the cells are re-arranged or extended, the change is automatically detected and the graph is re-generated.

Once the graph has been constructed, we apply a very simple, greedy approach in an attempt to route all required connections of a particular netlist. Each connection is called simply a net. Our simple algorithm works through all possible nets in an arbitrary initial sequence. When one net is routed, the routing resources (wires and switches) consumed to form that route are no longer available for nets still to be routed.

As one might suspect, this approach may not lead to a solution due to congestion. It is, of course, well known that routing problems such as these (which are referred to as graph Steiner Forest routing problems) are NP-hard, and require more sophisticated heuristics. As our priorities were on forming the elements of the basic system design, we did not extensively pursue more sophisticated algorithms. While we might suggest this as an "exercise for the reader", it will likely be re-examined as the scale of our demonstrations increases.

Fortunately, a rich base of research on these algorithms awaits us, and we will hope to have occasion to adapt them for the nuances of this novel architecture (such as "domain mapping" of nets into graph regions, analogous to graph coloring problems.).

3.2.6. Definition Of the Commands.

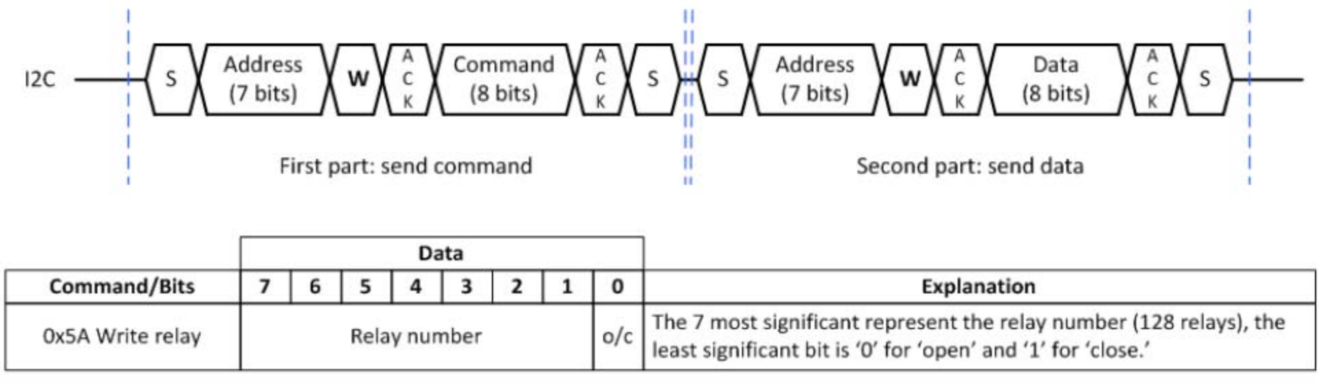
Using the I^{2}C protocol for the communication in the AWP, we defined each command for the control of the cell blocks. The commands are transferred from the cell management unit (CMU) to the cell blocks using the 8-bit data in the I^{2}C protocol. Thus, we have 256 possible commands. The commands that are already defined and implemented are shown in Table 2.

For each defined task, the process of transferring information changes. In agreement to the commands from Table 2, we have defined 4 types of commands. Depending on the task to be performed by the cell block, we have 4 types of commands. Every single type starts with a WRITE I^{2}C[9] communication. The following I^{2}C communications depend on the type of command. We describe each type in the next lines, including the timing diagram for each type.

1. *Type I*: This type of command requires the CMU to perform two WRITE I^{2}C communications. For example, in the case of the command 0×5A for opening/closing the delays, in the first I^{2}C communication the CMU sends the command to indicate that it will open/close a relay using a WRITE I^{2}C communication. In the second I^{2}C communication, the CMU writes the number of the relay to be opened/closed using a WRITE I^{2}C communication again. In Figure 18 we show the timing diagram for this type of command together with the characteristics for the 0×5A command.
2. *Type II*: This type of command requires the CMU to perform a two step I^{2}C communication: (i) send a WRITE and (ii) send a READ. For example, in the case of the command 0×44 for reading the ID of a cell block, the CMU first send a WRITE I^{2}C command to the cell block. Then, the CMU performs a READ I^{2}C command to read the ID of the cell block. In Figure 19 we show the timing diagram for this type of command together with the characteristics for the 0×44 − 0×48 commands.
3. *Type III:* This type of command requires the CMU to perform first a WRITE I^{2}C communication and then, the CMU needs to perform as many as READ I^{2}C communications are necessary. This type of command has been defined for reading the datasheet of the module connected to the cell block (0×5E command). For example, suppose that the datasheet of the module connected to the cell block has 9 bytes. First, the CMU must send the WRITE I^{2}C communication to the cell block. Then, the CMU sends a READ I^{2}C communication to receive the number of bytes (9 in this example) to be sent from the cell block to the CMU. Finally, the CMU sends 8 READ I^{2}C communications to continue reading the datasheet. In Figure 20 we show the timing diagram for this type of command together with the characteristics for the 0×5E command.
4. *Type IV:* This type of command requires the CMU to perform a three step I^{2}C communication: (i) send a WRITE (with the command to be performed), (ii) send a WRITE with the location of the internal address of the cell block's status to be read, and (iii) send a READ. For example, in the case of the command 0×5B for reading the status of one relay in the cell block, the CMU first sends a WRITE I^{2}C command to the cell block with the 0×5B command. Then, the CMU sends the number of the relay to know its status using a WRITE I^{2}C communication. Finally, the CMU performs a READ I^{2}C communication to let the cell block send the status of the requested relay. In Figure 21 we show the timing diagram for this type of command together with the characteristics for the 0×5B command.
5. *Type V:* This type of command was implemented to read the datasheet from the modules. For each cell with a module connected to it, the master will start reading the SPICE information from the module (that has been already stored in the cell). We are assuming that the maximum size for the SPICE information is 256 bytes. The master sends the command 0x60 (see Fig. 22), the memory address location of the SPICE information to be read, and then the cell sends the data.

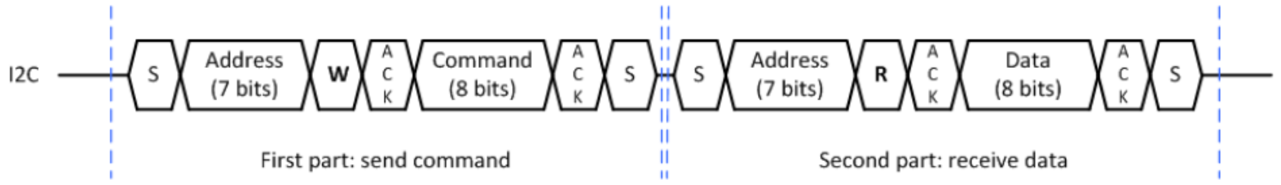
Table 2. Definition of commands for the communication between the CMU and the cell blocks.

Address						
Decimal	Hexadecimal	Binary	Command	Type	Data	Notes
0	00	00000000				First available command address
68	44	01000100	Read Cell ID	II	Cell ID	
69	45	01000101	Read North Neighbor ID	II	North ID	
70	46	01000110	Read East Neighbor ID	II	East ID	
71	47	01000111	Read South Neighbor ID	II	South ID	
72	48	01001000	Read West Neighbor ID	II	West ID	
73	49	01001001	Read Module ID byte	II	Module ID	
90	5A	01011010	Write relay	I	Open/close relay	Bits(7 to 1): number of relay. Bit(0) is ‘0’ when open and ‘1’ otherwise
91	5B	01011011	Read relay status	IV	Relay number	Write command, write relay number, read status.
94	5E	01011110	Read datasheet	III	Datasheet	Number of bytes, datasheet
255	FF	11111111				Last available command address.



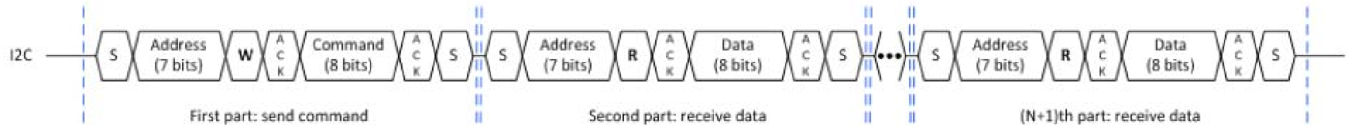
Command/Bits	Data 7	6	5	4	3	2	1	0	Explanation
0x5A Write relay	Relay number							o/c	The 7 most significant represent the relay number (128 relays), the least significant bit is ‘0’ for ‘open’ and ‘1’ for ‘close.’

Figure 18. Type I command.



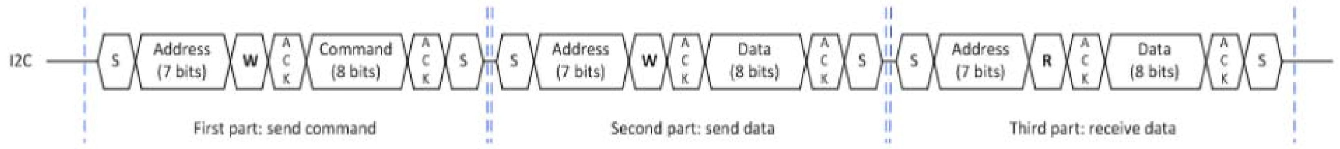
	Data								
Command/Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Explanation**
0x44 Read Cell ID	Cell ID								Read the ID of the cell.
0x45 Read North Neighbor ID	North Neighbor ID								Read the ID of the North neighbor of the cell. 0x01 means no cell.
0x46 Read East Neighbor ID	North East ID								Read the ID of the East neighbor of the cell. 0x01 means no cell.
0x47 Read South Neighbor ID	North South ID								Read the ID of the South neighbor of the cell. 0x01 means no cell.
0x48 Read West Neighbor ID	North West ID								Read the ID of the West neighbor of the cell. 0x01 means no cell.
0x48 Read Module ID	Module ID								Read the ID of the Module connected to the cell. 0x01 means no module.

Figure 19. Type II command.



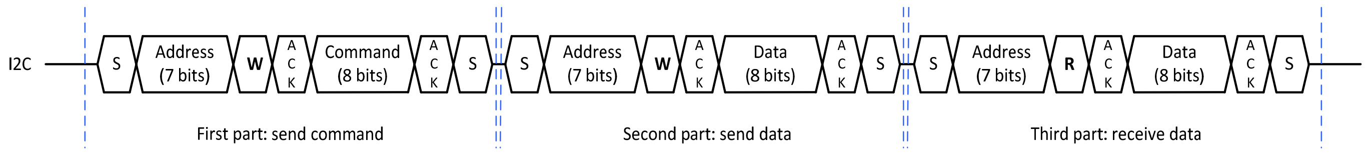
	Data								
Command/Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Explanation**
0x5E Read datasheet	First number of bytes to be sent, then the datasheet								When the datasheet needs to be read, the cell will send first the total number of bytes to be sent. Then, it will send the datasheet.

Figure 20. Type III command.



	Data								
Command/Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Explanation**
0x5B Read relay status	First the number of the relay to be read, then the status of the relay								Since the status the of the relay is only one bit, '0xFF' means 'close' and '0x00' means 'open.'

Figure 21. Type IV command.



	Data								
Command/Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Explanation**
0x60 Read datasheet word.	First the address of the datasheet memory to be read, then the value of the datasheet.								Maximum memory size: 256.

Figure 22. Type V command.

3.2.7. Command Line Interface.

The Command Line Interface (CLI) implements the Cell Management Unit on the computer side. It consists of a set of functions for communication with the Adaptive Wiring Panel (AWP). The main purposes of the interface are: (i) detect the AWP configuration (cells and modules arrangement), (ii) let the user set circuit connections, and (iii) make the connections and maintain the circuit once the connections have been set. The Graphic Unit Interface (GUI) runs on top of this Command Line Interface.

1. Detecting AWP configuration

   The process starts with the scanning of the I^{2}C bus looking for all possible peripherals (addresses from 1 to 127). When a peripheral (or Cell) responds, it provides the cell ID, the ID of its neighbors, the relative location of the neighbors to the cell (North, East, West or South), and the information of the module (if it exists) attached to the cell. The module information consists of its size (either 5×5 cm or 5×10 cm) and its orientation (0°, 90°, 180°, 270°).

   Table 3 shows an example with 6 cells interconnected. For each cell, the Cell ID is shown along with its neighbors' IDs. In addition, the table specifies whether a module is attached to a specific cell, and if it is, it displays the module information. The AWP configuration is the collection of the previous information for every available Cell.

Table 3. AWP Example.

Cell ID	Neighbors				Module attached
	North	South	East	West	
1A	5B	0F	9E	1D	Yes, 5x5, 90°
5B		1A			No
1D		3C	1A		No
9E				1A	Yes, 5x10, 0°
0F	1A			3C	No
3C	1D		0F		No

Based on the existing AWP configuration, the software routine creates the Cell Array, the Module Array, and an undirected Graph that specifies all terminal connections within the Cell Array. These 3 layers of information and explained in the next lines.

- The Cell Array is a matrix that represents the arrangement of the cells with respect to each other. This information is vital in order to build the Graph.
- The Module array is a structure that represents the arrangement of the Modules and the information of each module; this information includes the module orientation, the list of components, and the location of each component within the module. A module can span several cells.
- The Graph with the terminal connections is an undirected graph that specifies connections between terminals of the same cell (through relays), and connections between terminals of different cells (thru wires). The connections between terminals of the same Cell are predefined information. The Cell Array provides us with the connections between terminals of different cell.

Figure 23 shows these 3 layers of information for the example of Table 3. Note how the 5×5 cm module spans 2 cells since it is rotated 90°. Also, the 5×10 cm module attached to Cell 9E only spans 1 Cell because Cell 9E has no neighbor to the East.

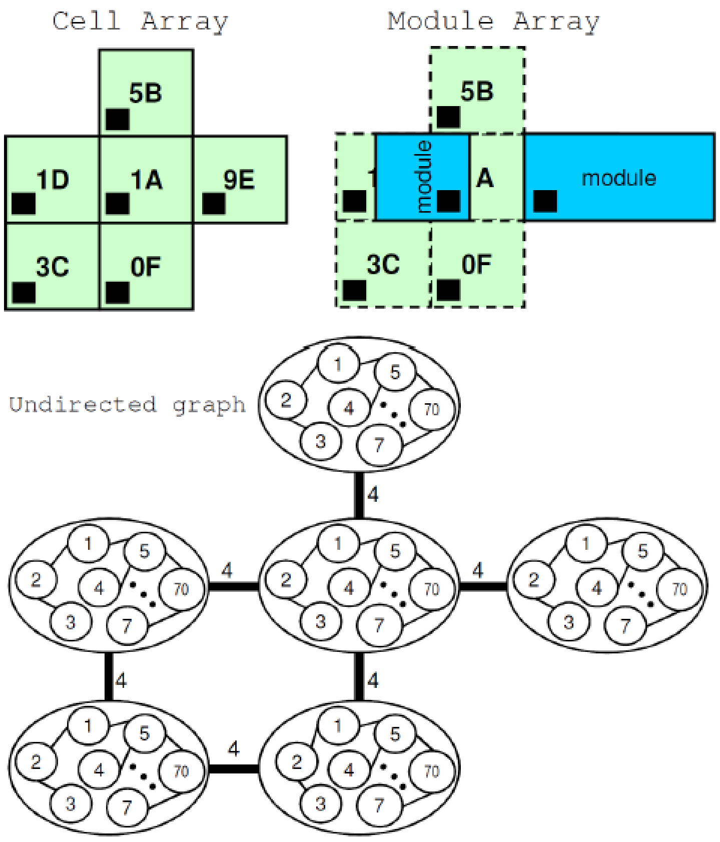


Figure 23. AWP example with 6 cells and 2 modules.

2. Setting circuit connections

   The user is given information about the Module Array. The circuit connections are set at this level. Each circuit connection is defined as a pair connecting a terminal of one component with a terminal of another component (or maybe the same component).

3. Make the connections and maintain the circuit once connections have been set

   The connection set is made of ‘N’ pairs. Each connection is defined as a pair. A pair is mapped as two nodes in the undirected graph.

   The basic heuristic algorithm to make the connections is summarized in Figure 24. An ordering of the ‘N’ connection pairs is generated (using the Johnson-Trotter algorithm[10]). We apply the shortest path to each connection pair. If a connection is not possible, we try with a different ordering. This process is repeated until a solution is found (exiting the algorithm with ‘Success’), or until all the orderings (N!) are exhausted, exit the algorithm with ‘Failure’.

   Usually, connections are not possible because the previous connections utilized nodes and edges, and as a result, a new connection pair might be left with no path that links them.

   Once the circuit is set, the system enters a loop in which it keeps the connections in response to an AWP modification (e.g., cells or modules change location, new cells or modules are installed). The user can always modify the connections or add more connections. Each time one of the previous events occurs, a new graph is generated, over which the algorithm is run again.

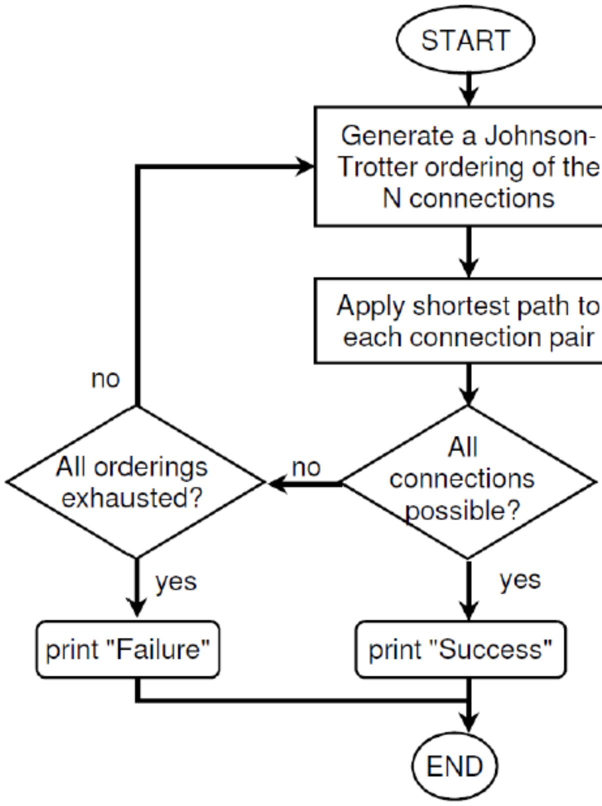

START
Generate a Johnson-Trotter ordering of the N connections
Apply shortest path to each connection pair
All connections possible?
no
All orderings exhausted?
no
yes
yes
print "Failure"
print "Success"
END

Figure 24. Algorithm to connect 'N' pairs.

4. Other functions

   The Command-Line Interface also provides a set of low level functions for debugging purposes: unconnect circuit, writing/reading data to/from a cell, reading data from a module, writing information (Spice datasheet) on a module.

3.2.8. Graphical User Interface (GUI) for the AWP.

The GUI was developed to ease the usage of the AWP system, releasing the necessary to remember different commands. It is developed based on the GTK+ library.16 The main function of the GUI is to keep updating the visual display when an operation executed on the panel, recognize user's intention to make connections and keeps the connections when operations are carried out.

The operations include changing the relative positions of the cells, moving the module boards over cells, rotating the module boards, programming the module boards, etc. When any operation is carried out, the acknowledge of the operation is displayed on three different layers of the GUI: (i) cell array layer (see Fig. 25), used to display the relative positions of the cells, (ii) module layer (see Fig. 26), to show the rotation and relative positions of the module boards, and (iii) circuit layer (see Fig. 27) to draw the symbols of the components.

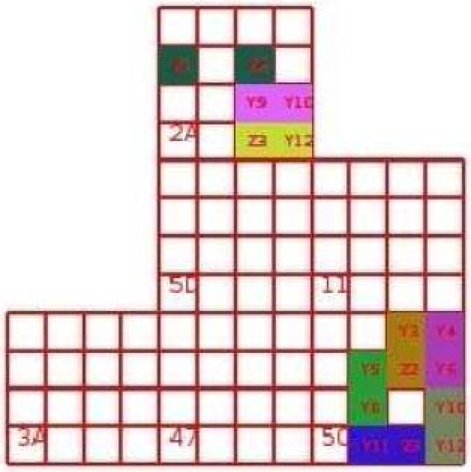
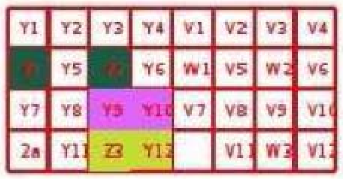


Figure 25. Cell array layer.

Once the user decides to make a connection, one can do some clicking on the module layer to make connections between pins. Or he can zoom and move the components on the circuit layer to make room for connections. After the components are positioned and scaled in a proper way, the user can just make connections by clicking the terminals of the components. A simple rectangle will show up to tell the user they have made a correct operation. After the user has finished the connection, a confirmation has to be made to let the program talk to the cells to link those connections.



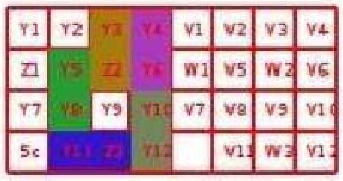


Figure 26. Module Layer.

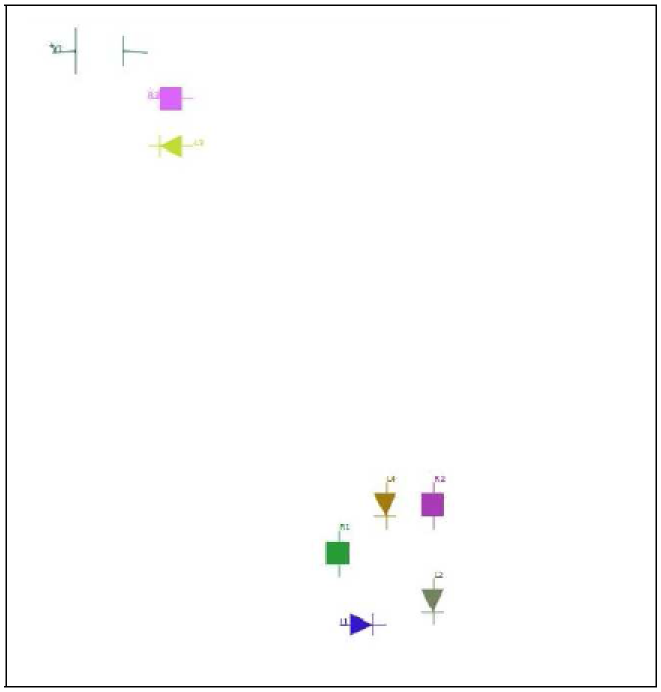


Figure 27. Circuit Layer.

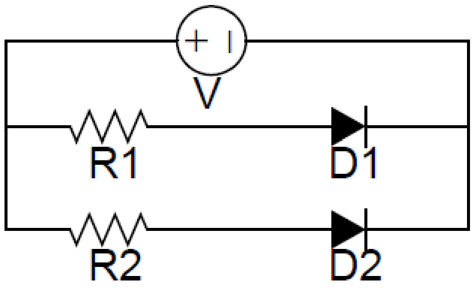
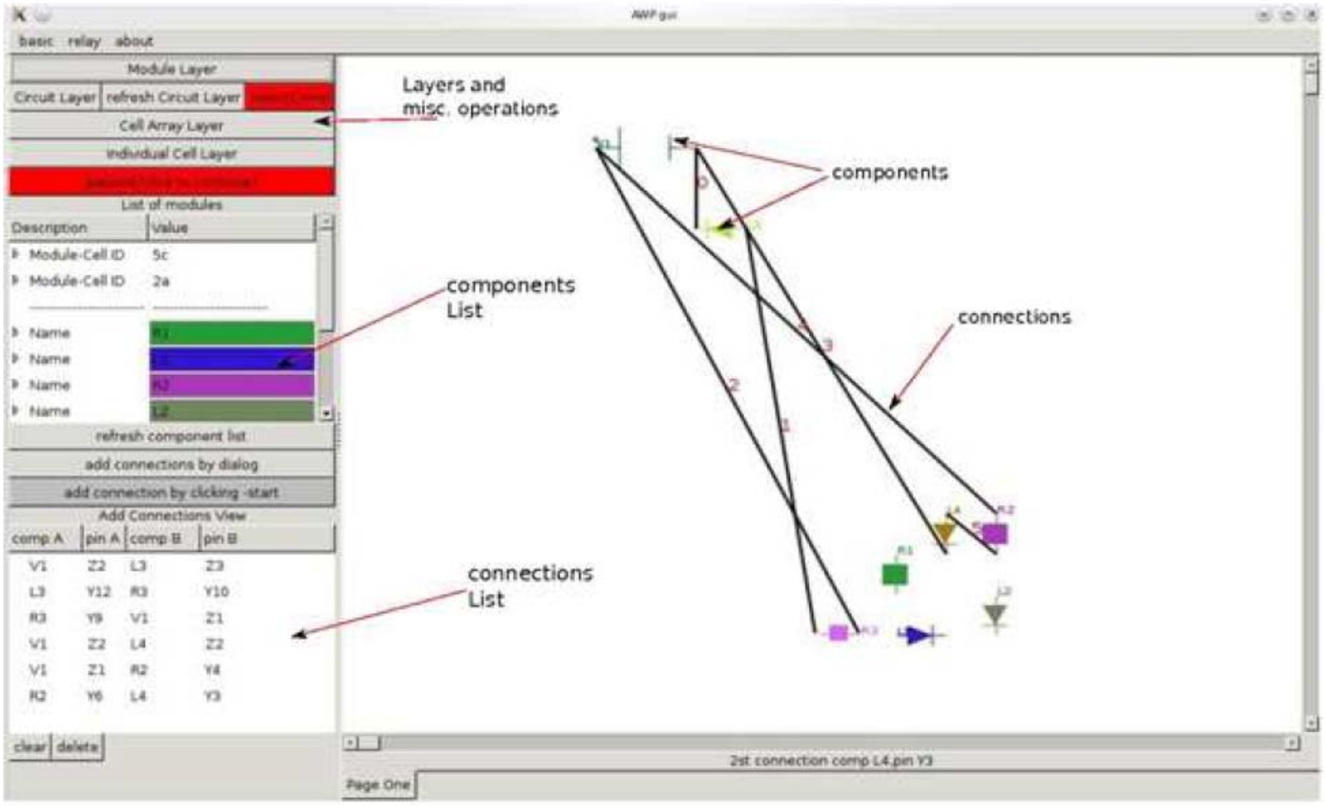


Figure 28. Circuit to be implemented.

For example, if we want to show a circuit as the one shown in Figure 28, first, the user can zoom and move the components for a better visualization. Then click and make connections as shown in Figure 29. Follow the number above the line which indicates the sequence of connection operations.



Description	Value
Module-Cell ID	5c
Module-Cell ID	2a
Name	R1
Name	[illegible]
Name	R2
Name	L2

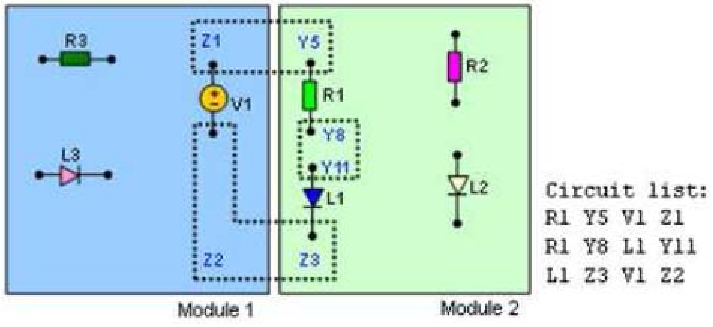
Add Connections View

comp A	pin A	comp B	pin B
V1	Z2	L3	Z3
L3	Y12	R3	Y10
R3	Y9	V1	Z1
V1	Z2	L4	Z2
V1	Z1	R2	Y4
R2	Y6	L4	Y3

Figure 29. Window overview and connections made for the 2-LED circuit.

3.2.9. Status of Demonstration.

Our prototype, a partial panel containing six cells and two modules, has demonstrated all of the elements of the AWP described in this section. Module 1 has been prototyped as a "compound" consisting of a battery source (V1), a resistor (R3) and a LED (L3). Module 2, also a "compound", has two resistors (R1 and R2) and two LEDs (L1 and L2). Simple circuits of the form shown in Figure 30 can be composed, in which subsets of modules can be connected.



```
Circuit list:
R1 Y5 V1 Z1
R1 Y8 L1 Y11
L1 Z3 V1 Z2
```

Figure 30. Example of a circuit to be connected.

Even these simple demonstrations have considerable underlying complexity, as each cell contains a dedicated processor, internal wiring and 70 relays to implement local connections. The modules also have internal microprocessors to "explain" modules to the adaptive system. Our eventual demonstration will contain 64 cells, resulting in nearly 4500 relays. To manage this complexity, tools will be necessary. Figure 31 provides a snapshot of the graphical user interface (GUI) software developed for the cell management unit.

We have successfully created a GUI that can manage the creation of circuits on the AWP. Once a circuit is set-up, the user can re-arrange either the cell units or the modules. The AWP will look for the new locations of the components previously defined and adapt the routing to connect the circuit.

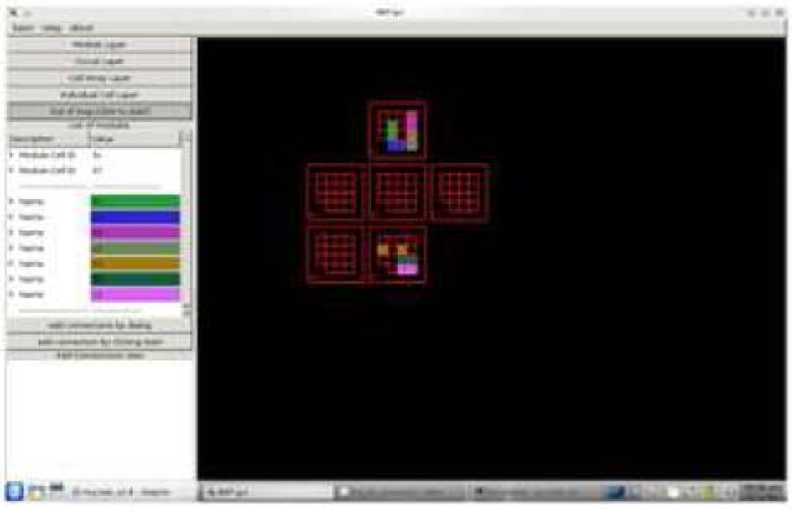


Figure 31. GUI snapshot of the software implemented for the Cell Management Unit.

3.3. Second AWP Prototype Design and Implementation.

After the example in Section 3.2.1 turned out to be successful, we developed a new Adaptive Wiring Panel Prototype. The complete AWP (48 cells) was verified to work with both the command line interface and Graphical User Interface (GUI). This prototype (called AWP 2nd Prototype) is fully operational. The hardware and software has been tested to work with several circuit examples.

The current prototype uses 48 Cell Units arranged in a 6x8 configuration. Figure 31 shows a Cell Unit. The AWP 2nd Prototype consists of a Panel that can hold the 48 cells. Each cell connects to the panel via a right-angle connector. The cells are interconnected via ribbon cables, facilitating the plugging/unplugging of the cells from the panel.

The components are provided via a Module Unit. Twelve (12) modules are available (6 of size 5cmx5cm and 6 of size 5cmx10cm) that can be programmed via I^2C with any circuit. Figure 32 shows a picture of the two module types with some components attached.

Figure 34 depicts the AWP 2nd Prototype. Figure 34(c) shows the underside of the AWP. Figure 34(d) shows a front view of the panel. Modules are plugged in and out on this panel.

Figure 35 shows an example where the AWP is operating. Figures 35(b) and 35(c) show how the command-line and GUI successfully detect the 48 cells. Figure 35(d) shows a picture of the panel with 2 modules plugged in, and a working circuit. The components of the modules were successfully detected as can be seen in Figure 35(c).

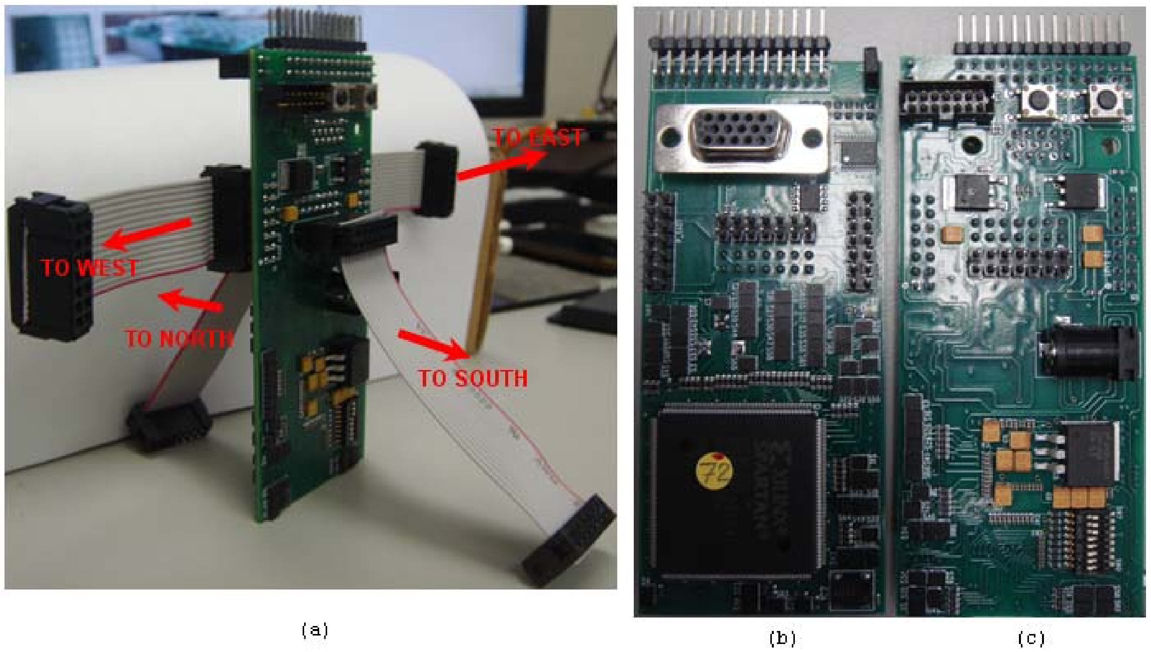


Figure 32. Cell Unit

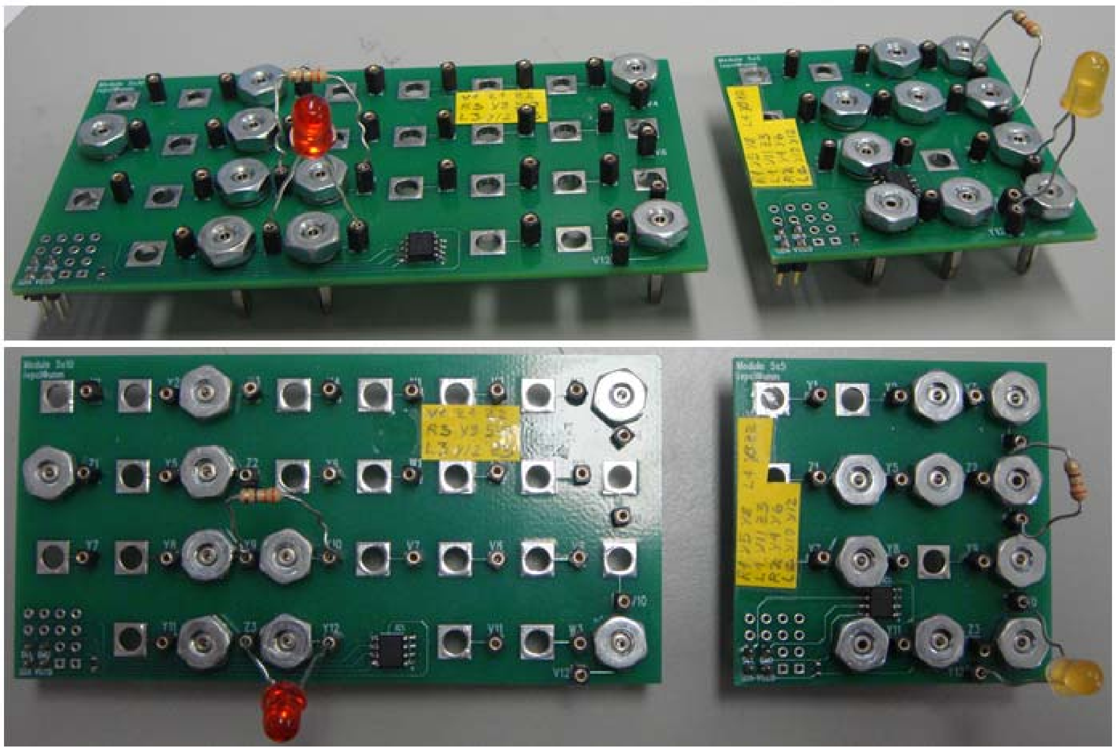


Figure 33. Module Boards (5cmx5cm and 5cmx10cm)

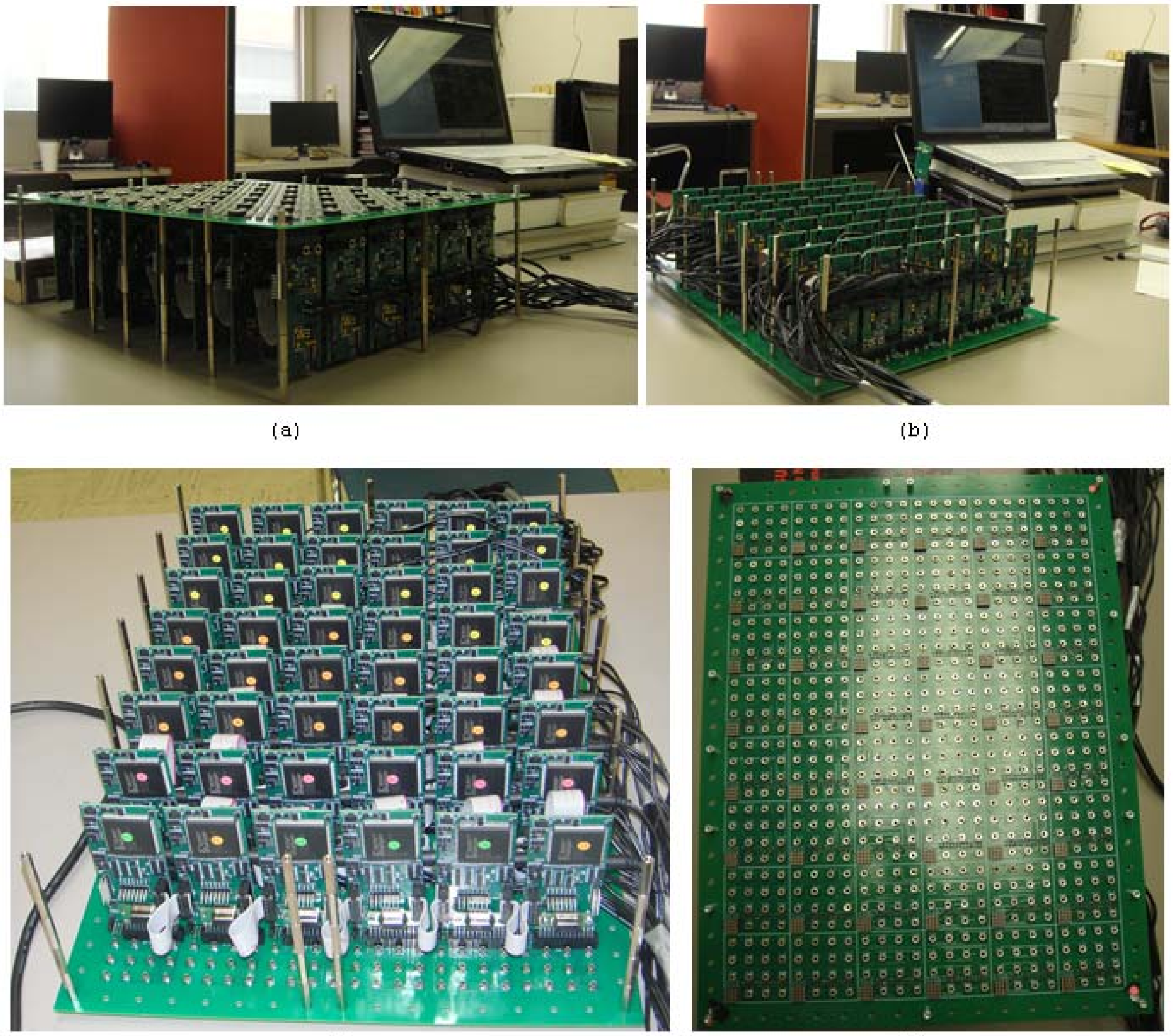
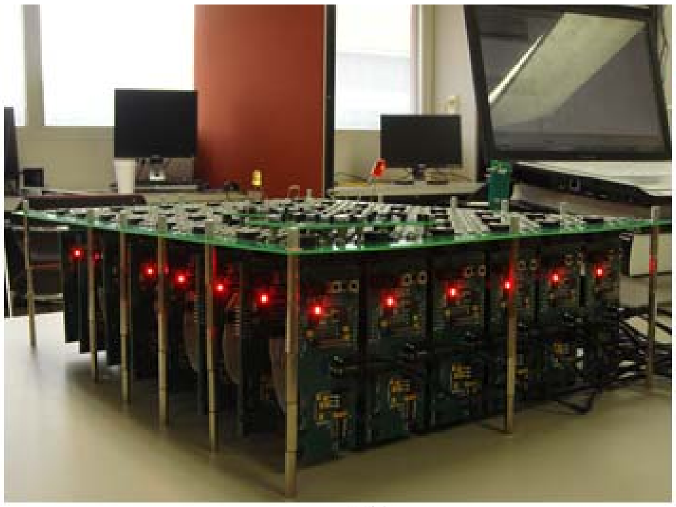
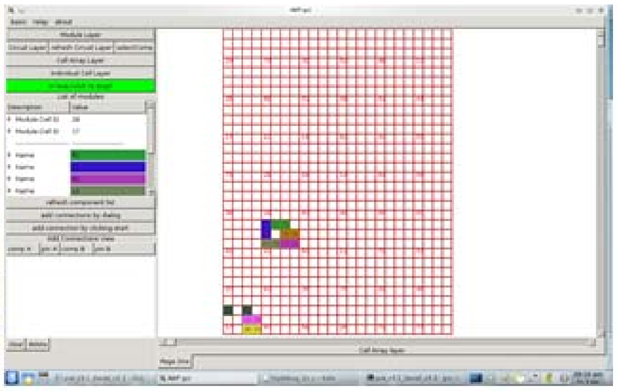
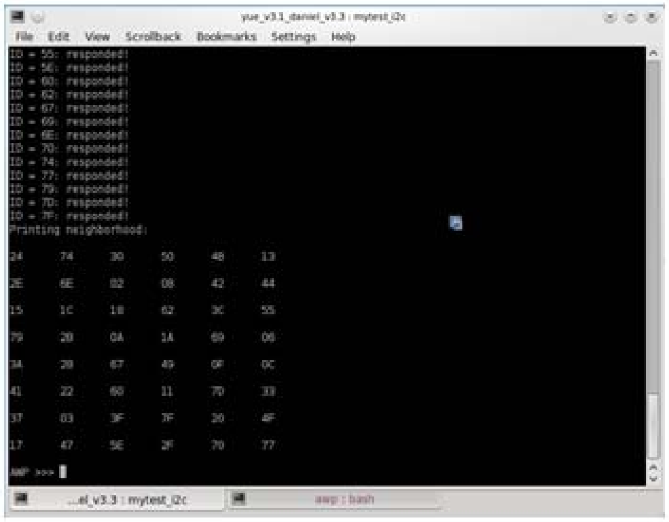
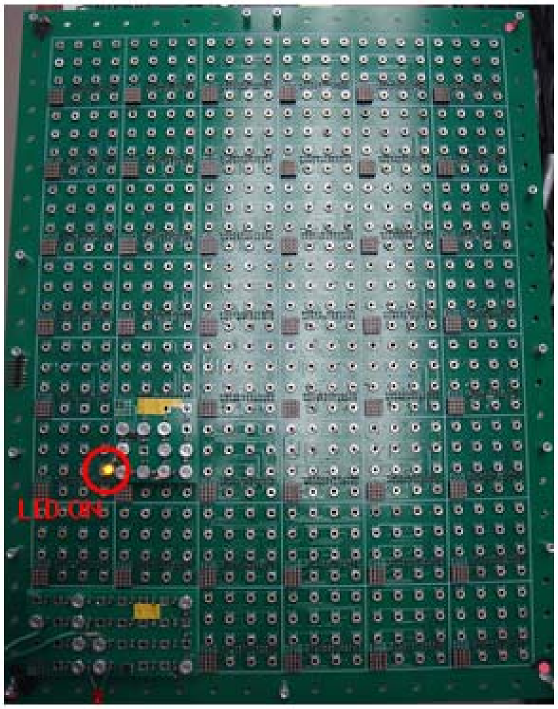


Figure 34. Complete AWP 2nd Prototype.


(a)


(c)


```
ID = 55: responded!
ID = 5E: responded!
ID = 60: responded!
ID = 62: responded!
ID = 67: responded!
ID = 69: responded!
ID = 6E: responded!
ID = 70: responded!
ID = 74: responded!
ID = 77: responded!
ID = 79: responded!
ID = 7D: responded!
ID = 7F: responded!
Printing neighborhood:

24      74      30      50      48      13

2E      6E      02      08      42      44

15      1C      18      62      3C      55

79      20      0A      1A      69      06

34      28      67      49      0F      0C

41      22      60      11      7D      33

37      03      3F      7F      20      4F

17      47      5E      2F      70      77

AWP >>> 
```
(b)



(d)

Figure 35. Complete AWP 2nd Prototype in operation.

3.3.1. Power Consumption of the AWP.

The power consumption for each cell has been measured with the ES-687 clamp meter. The idle power (no relay activated) for a cell is about 0.42W, while the power consumption per relay is about 85mW.

Figure 36(a) shows the current consumption (mA) per cell for an increasing number of activated relays. Figure 36(b) shows the power consumption (mW) per cell. Figure 36(c) shows the power

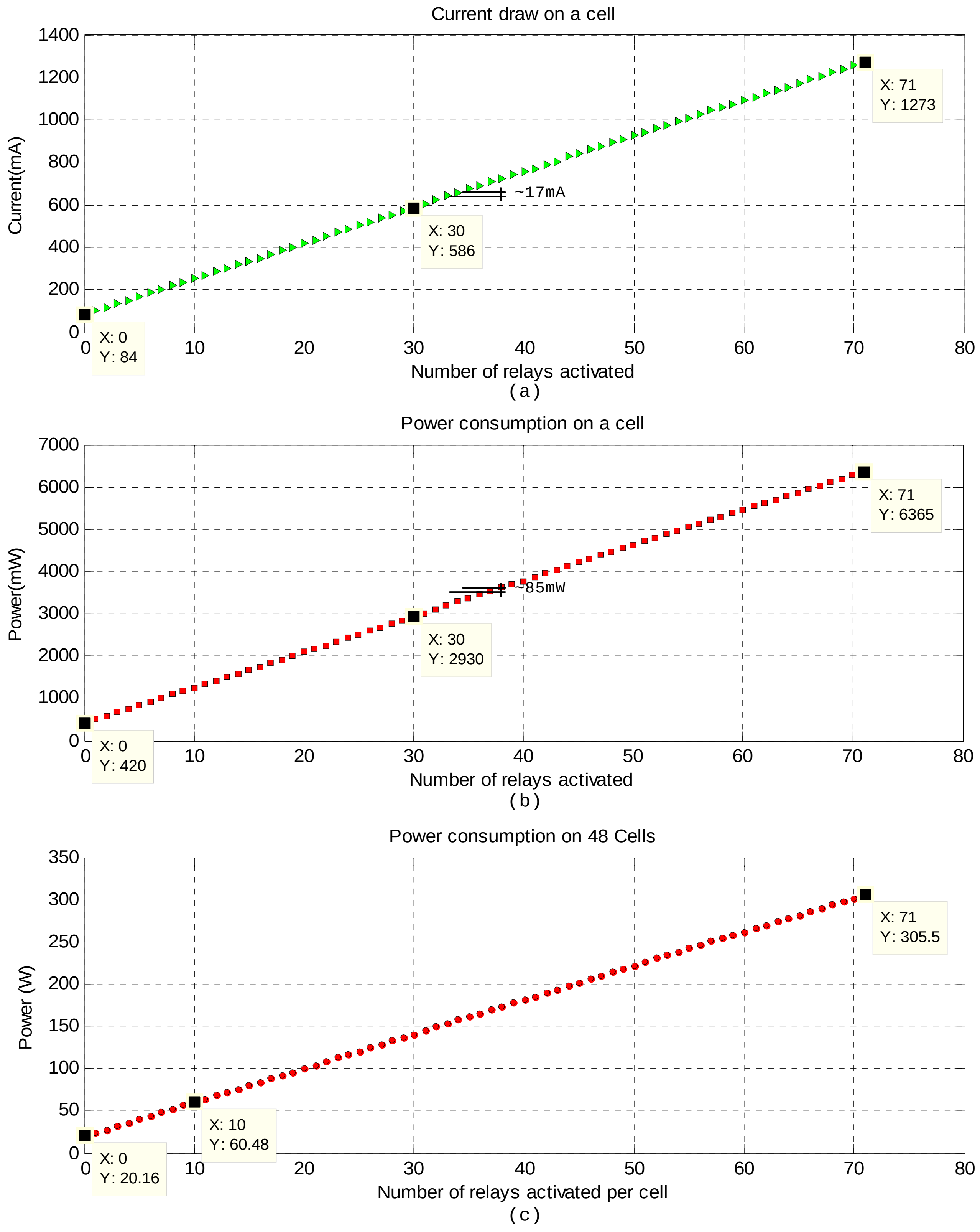


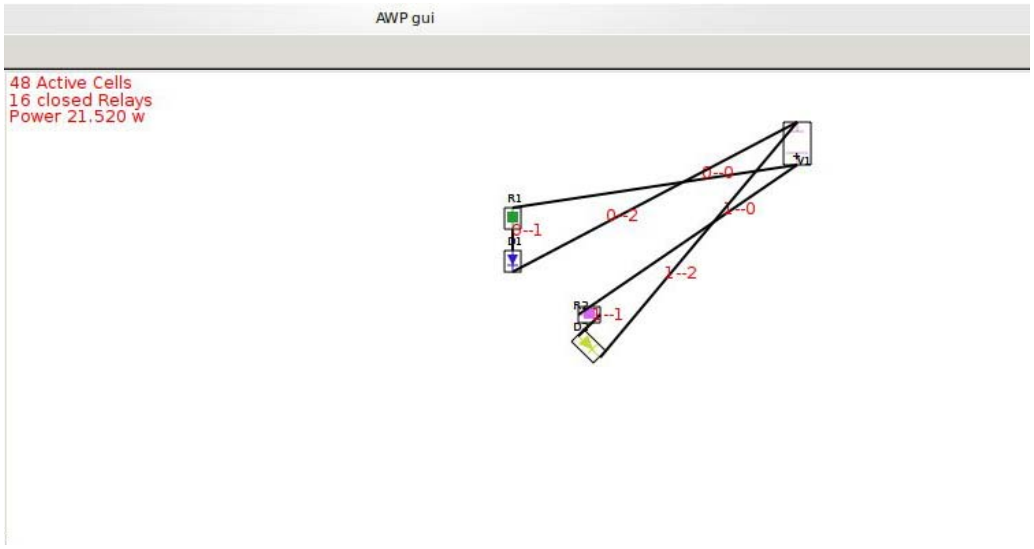
Figure 36. Power consumption for the AWP.

consumption for 48 cells. We can see that the idle power consumption of the AWP is about 20.16W, and the maximum possible power consumption is about 305.5W. However, in practice, we expect that the average number of relays activated per cell will not exceed 10. In this case, the AWP draws about 60.48 W.

With data from Figure 36, we can estimate the power consumption of the AWP in real time. The Graphical User Interface includes real-time power estimation based on the number of active cells and the number of closed relays.

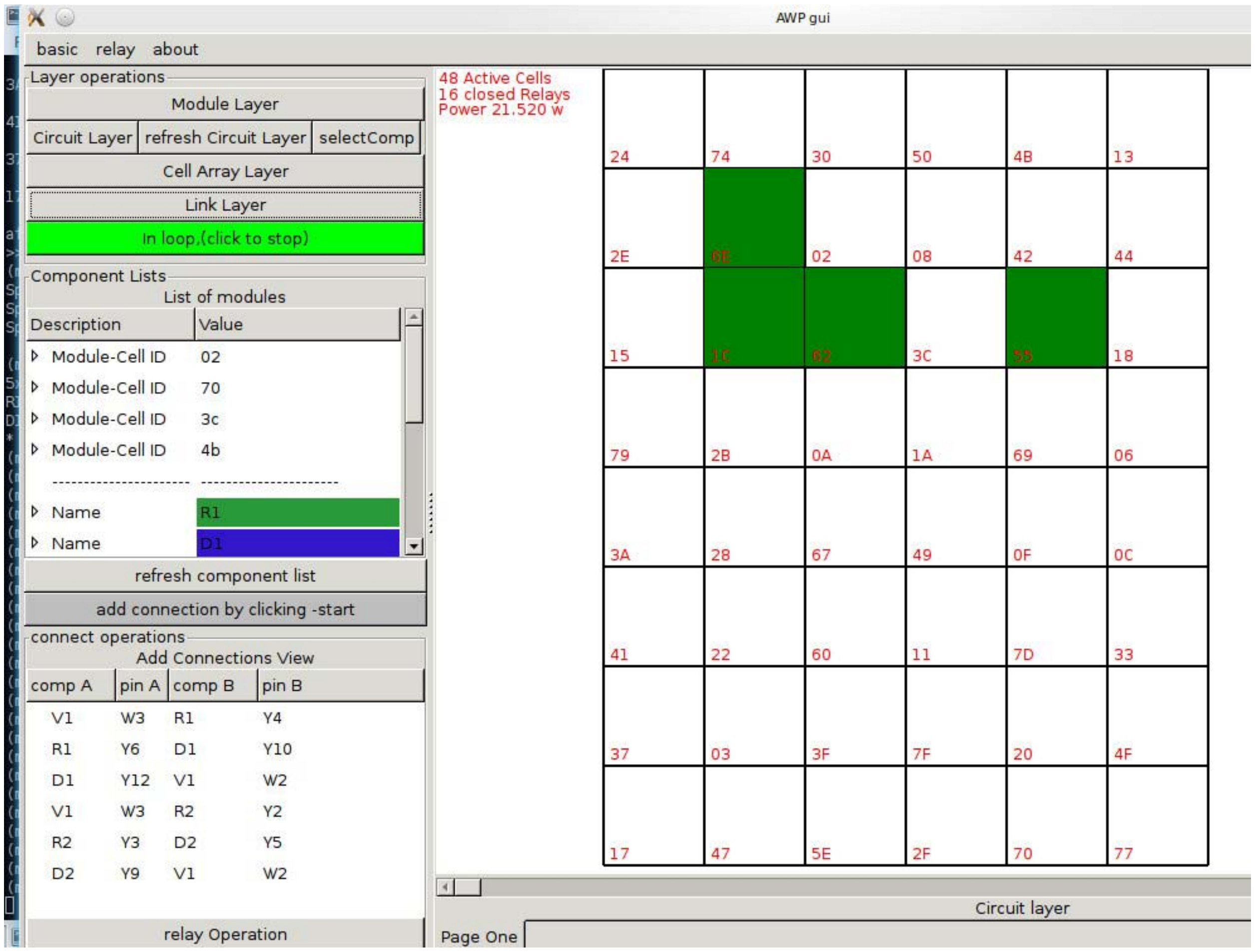
3.3.2. GUI Improvements.

A new layer, called 'Link Layer' has been added to the Graphical User Interface (GUI). This layer shows the relays that are currently closed and the cells. If there are relays closed inside a cell, the cell turns green. By double-clicking the specific cell, the user can see the cell in detail with the condition of the relays (closed/open). Figure 37, 38, and 39 show an example. Figure 37 shows an example circuit array connected in the system. Figure 38 shows the cells in which their relays are closed. Figure 39 shows the relays inside cell 62, the highlighted rectangle denotes a closed relay.



48 Active Cells
16 closed Relays
Power 21.520 w

Figure 37. Two (2) circuits in AWP system.



```
48 Active Cells
16 closed Relays
Power 21.520 w
```

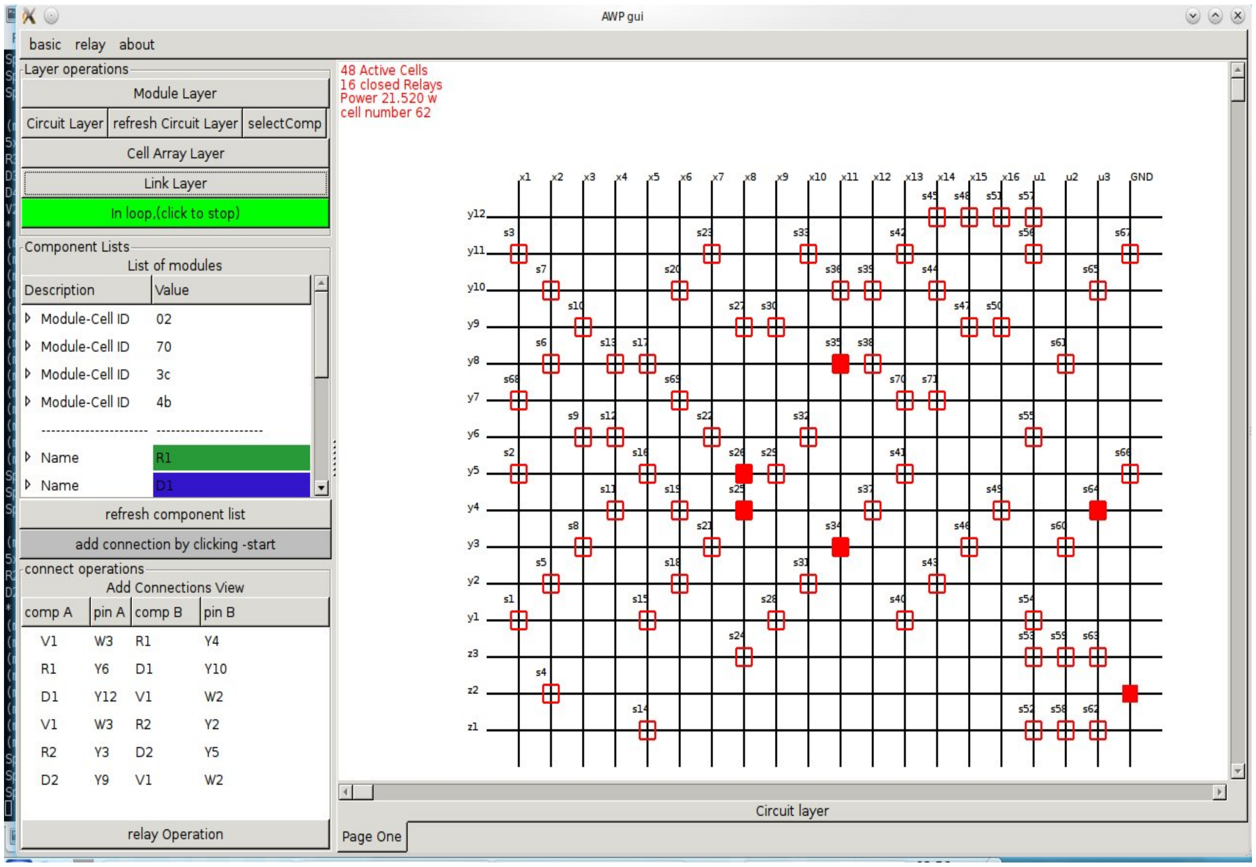
List of modules

Description	Value
Module-Cell ID	02
Module-Cell ID	70
Module-Cell ID	3c
Module-Cell ID	4b
---------------------	---------------------
Name	R1
Name	D1

Add Connections View

comp A	pin A	comp B	pin B
V1	W3	R1	Y4
R1	Y6	D1	Y10
D1	Y12	V1	W2
V1	W3	R2	Y2
R2	Y3	D2	Y5
D2	Y9	V1	W2

Figure 38. Link layer for example circuit in Figure 37.



48 Active Cells
16 closed Relays
Power 21.520 w
cell number 62

Description	Value
Module-Cell ID	02
Module-Cell ID	70
Module-Cell ID	3c
Module-Cell ID	4b
---------------------	---------------------
Name	R1
Name	D1

comp A	pin A	comp B	pin B
V1	W3	R1	Y4
R1	Y6	D1	Y10
D1	Y12	V1	W2
V1	W3	R2	Y2
R2	Y3	D2	Y5
D2	Y9	V1	W2

Figure 39. Inner relays for circuit in Figure 37

3.3.3. Gesture Detection Interface.

The AWP Panel PCB poses some challenges with respect to the original PCB in which we test the gesture detection algorithms (Fig. 40). The challenges include: i) Stronger light reflections, ii) the green color of the AWP Panel PCB, and iii) The fact that the AWP Panel PCB (Fig. 41) is larger, which leads to smaller resolution for each pin of the AWP Panel.

As a result, the Software Interface for the AWP 2nd Prototype, based on OpenCV and GTK+ libraries, requires a robust algorithm to compensate for the problems mentioned above.

Figure 42 shows the start of the program where 3 windows can be seen:

- Screen display: This window shows the frame captured by the camera. It also shows information worked out by the program like grid, corner position, and finger positions.
- Foreground display: This window shows the results of the fore-background algorithm (a robust and simple method to detect the finger tips). First, a background frame is stored for later reference. The foreground frame is created by subtracting the current frame from the background frame. If the difference exceeds tolerance, it is white colored. The program first detects the four corners of the frame to determine the finger position. For example, if a

change is detected in the bottom line, it means that the finger is pointed up; thus, the finger tip is scanned in a top to down, left to right fashion. If a change is detected in the left border, then the finger tip is scanned in a right to left, top to down fashion.

- Control Panel: It provides a set of buttons and boxes to configure the procedure parameters. Also, it features a tree view widget to show the connections set by fingers.

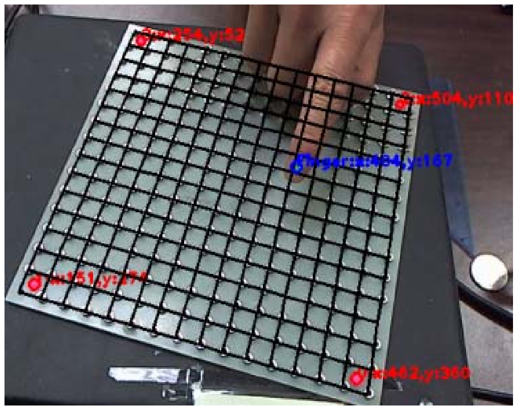


Figure 40. Original PCB Board for testing purposes

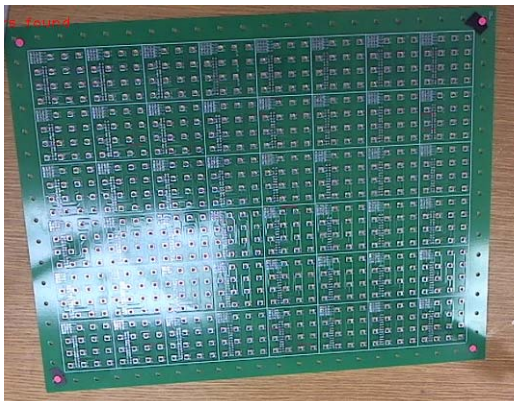
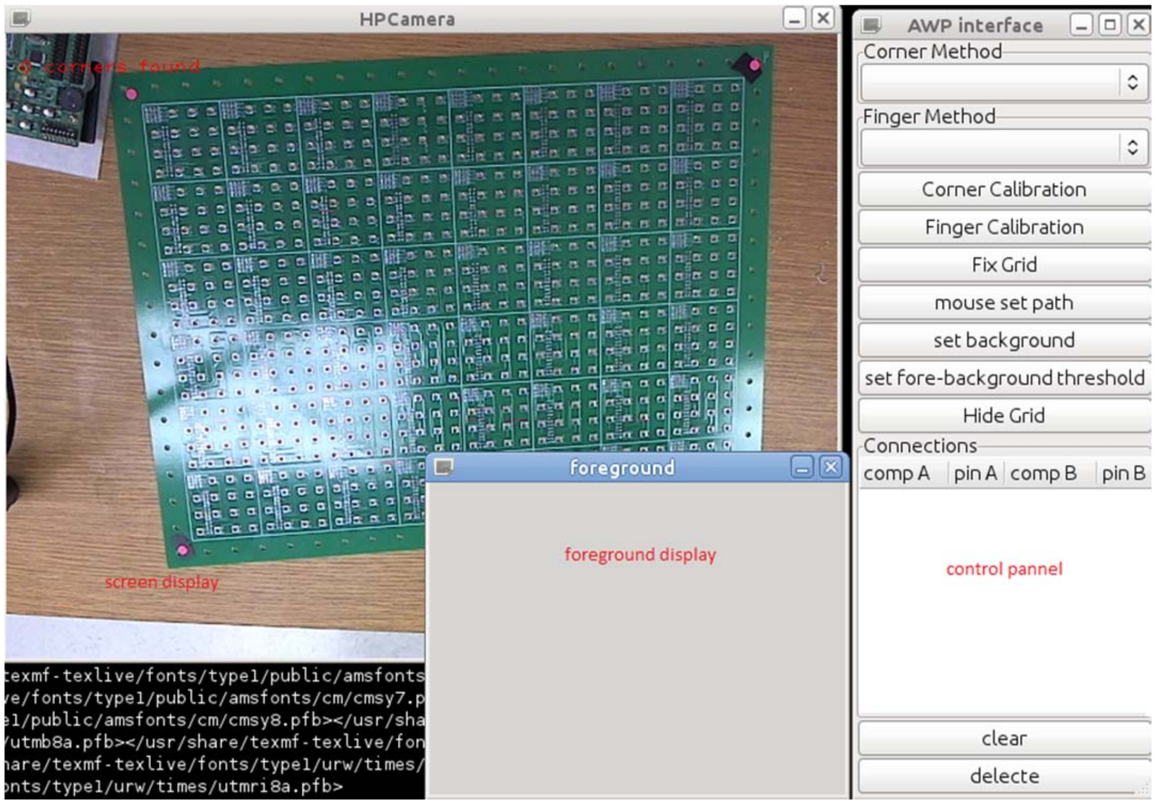


Figure 41. Actual PCB Board (AWP Panel)



```
texmf-texlive/fonts/type1/public/amsfonts
ve/fonts/type1/public/amsfonts/cm/cmsy7.p
e1/public/amsfonts/cm/cmsy8.pfb></usr/sha
/utmb8a.pfb></usr/share/texmf-texlive/fon
hare/texmf-texlive/fonts/type1/urw/times/
onts/type1/urw/times/utmri8a.pfb>
```

Figure 42. Graphic Interface at the Start of the Program

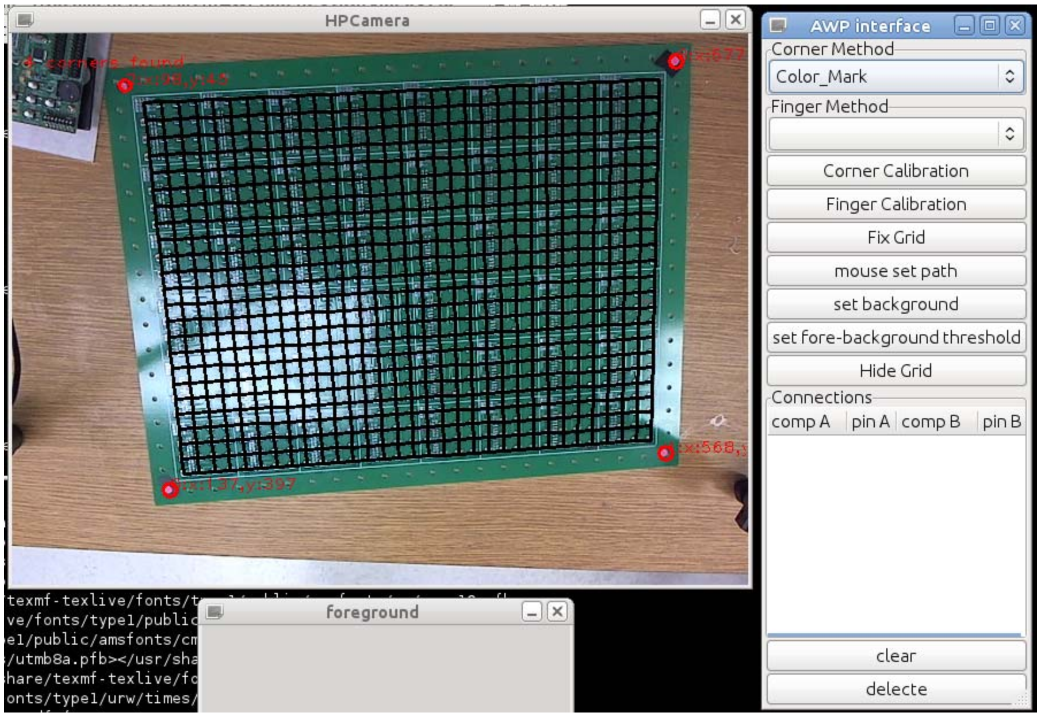
After the program is started, the corners are set up, so the grid can be generated according to the corners' position information. This is carried out automatically (see Fig. 43) or set by the user (with the fingers).

To detect connections created by finger gestures, the program has an internal counter to detect the finger motion pattern. If the finger stays on one pin for one second, the program determines that the user wants to make connections on this pin. A connection is set up by two pins, and the connection will show up in the tree view widget on the control panel. Users can also clear the connections and delete a certain connection.

This software interface that let us create a list of connections needs to integrate to the AWP GUI, so that the connections can actually be set.

3.3.4. AWP Publications.

The results of the work on the Adaptive Wiring Panel have been also published in: [2], [3]. In addition, results of Section 3.2.2 were submitted for publication to the *AIAA Journal of Aerospace Computing, Information and Communication*, and it is currently under review.



```
texmf-texlive/fonts/t
ve/fonts/type1/public
el/public/amsfonts/cm
/utmb8a.pfb></usr/sha
hare/texmf-texlive/fo
onts/type1/urw/times/
```

Figure 43. Automatic corner detection

4.0. Conclusions.

4.1. AFRL-UNM HERC Prototype.

The AFRL-UNM HERC was conceived with the objective of exploring the advantages and disadvantages of a high-end reconfigurable computer, built using FPGAs as their main component. This exploration was constrained by limiting the system to applications involving space vehicles, and therefore, special attention was given to issues such as the system's size and power consumption, as well as the power consumed by scaled versions of the system.

An architecture satisfying these needs was proposed. Some of its features were implemented as a first prototype. Modularity and scalability of the system were achieved by interconnecting Basic Modules in a unique mesh topology. Ethernet and RocketIO were explored as alternatives for inter-modules communications, and a custom communication link was proposed and explored for inter-processor communications in a single module.

The basic module was designed and implemented with the objective of obtaining the fastest possible single-processor system hosted by a FPGA. Therefore, high-speed DDR2 memories were used in this design, a high-speed clock distribution and supporting systems implemented, and high-speed communication modules were included. The basic module was implemented with high-speed connectors (currently used as low-speed connectors), and a System Ace peripherals intended to provide HERC developments with expansion support and a high-volume storage system.

Other features were also included in the basic module. They are low-speed Analog to Digital Converters (ADCs), an USB-2 host peripheral intended to provide peripheral expandability to the system, and a low-speed network, common to all Basic Modules (BMs) in the HERC. Although partially implemented in the prototype of the HERC, these features allow future users to research more complex systems in this reconfigurable platform.

The aforementioned hardware, and its power distribution system were implemented in a 150mmx200mm, 20 layer printed circuit board. This PCB was especially designed to provide high-speed, controlled impedance traces in several interconnects, and to be area efficient. Two BMs, hosted by a high-speed data and a high-capacity power backplane, were built as a prototype of the proposed HERC.

To support the communication between the multiple processors in the HERC, and to facilitate applications developing and execution, a version of the Linux operating system was ported to the BM. This OS was enhanced with customized network drivers, and a message passing communications engine based on OpenMPI was implemented. This software platform was chosen to allow future users of the AFRL-UNM HERC to be able to create hardware and software applications for this system in commonly available platforms, without the need of having their own HERC available. Additionally, special hardware was created to collaborate with this software platform, supporting the internal and external communication links in the system.

4.2. Adaptive Wiring Panel.

We have described an unusual architecture to interconnect (in principle) arbitrary electronic components together using a programmable manifold. The adaptive wiring concept, in one way of thinking, is an extension of the ideas of FPGA routing. The ideas clearly demonstrate that a considerable investment in overhead is required, even to do simple things, like turn on a light bulb. Similarly, FPGAs also come with considerable overhead. For a million gate systems, the overhead is often acceptable. The same level of overhead for a 10-gate FPGA, however, would be considered profligate. As such, the power and utility of AWM will likely become more appreciated with larger scale systems (DeHon described a similar phenomenon with Minnick's work on cut-point cellular arrays in the 1960s, with a briefcase-sized system required to demonstrate a few tens of gates equivalent in expressive capacity[11]). The technology implications for fully adaptive wiring are potentially profound. Systems can be formed more quickly, resilient, and flexibly. The benefits come at a price, namely that of excess overhead, the need to make components (modules) "smart", and the need to have tools, such as a synthesis engine, to manage the complexity of the dynamic wiring. Our initial work has progressed to a demonstration system with 48 cells. At this scale, we expect to find no technological surprises, but expect to uncover new insights of application potential and learn how to better cope with overhead. We expect to learn to balance global and local considerations (should tiles determine their own local routes?) and to optimize better the balance of switch and wire resources. Is there a benefit in extending these concepts to three dimensions, replacing the notions of smart tiles with "smart cubes"? Can we finally achieve breakthroughs in micro-electromechanical systems to allow us to economically implement a hundred thousand relays in integrated form instead of the painfully tedious discrete implementations of today? In the future, we may, instead of bringing modules into adaptive panels, find instead it is better to simply make all modules adaptive in their own bundle of configurable wires, forming perhaps the ultimate form of configurable system.

5.0. Recommendations.

5.1. AFRL-UNM HERC Prototype.

It was found that the performance of the BM, when benchmarked as a single-processor system, was in between the performance of an Intel Pentium microprocessor running at a clock frequency of 133 MHz, and an Intel Pentium II running at a clock frequency of 300 MHz. This comparison is an average of the results obtained of the benchmarking on the characteristics of systems built in the BM. The obtained performance, and the BM multiprocessor capabilities, places the BM in a high ranking when compared to similar embedded systems. This performance still can be improved if changes are made in the memory controller.

The memory controller supporting the DDR2 memories on the BM imposed a serious bottleneck on the system. This is a memory controller available from Xilinx that interfaces the PLB bus with the DDR2 memory DIMM on the HERC. It was designed to only interface with a 100 MHz PLB bus despite the memory's operating frequency. Additionally, it requires six clock lines for its operation, consuming FPGA clocking resources in a clock domain, enough to prevent an easy routing of other clocks with higher speeds. Therefore, it also prevents the microprocessor for being clocked at higher frequencies, or to use a higher frequency to clock the PLB bus. For this reason it was not possible to reach the highest frequency the system was designed to support, and the design, or the purchase, of a more appropriate memory controller is recommended.

The results of benchmarking the communication links show that the AFRL-UNM HERC with processors hosted by an Ethernet network, exhibited a communication performance similar to clusters in other network platforms, such as in a cluster of Sun SparcStations, over 10Mbps Ethernet, or in a cluster of DEC 3000/300 workstations on a FDDI network. It is expected that future improvements in the performance of the BM single-processor characteristics, will substantially improve the execution rate of multiprocessor applications, due to a better utilization of the AFRL-UNM HERC network.

The FPGA resource consumption of the systems implemented for the HERC operation indicate that a Virtex-4 FX 40 is the smallest FPGA where a single-processor BM system can be implemented, and that the Virtex-4 FX 60 is the smallest one where a multiprocessor implementation can be achieved. The prototype of the AFRL-UNM HERC built was implemented in a Virtex-4 FX 100 FPGA, which allowed building and studying complex BM implementations, while saving resources for future enhancements.

It was also found that in a HERC where its BMs are interconnected by Ethernet links, up to ten of these links should be implemented in a Virtex-4 FX 100 to avoid depleting its resources, and at the same time, to leave enough space for the user's hardware applications. In a HERC featuring RocketIO based communication links, fourteen links can be implemented before exhausting the global buffer resources in the FPGA.

The power consumption analysis allowed obtaining important conclusions regarding the hardware characteristics that a HERC such as the proposed one should have. It was found that a HERC made of Ethernet based inter board communication links consumed less power than its equivalent made of Rocket IO links. However, as aforementioned, it was also found that resource utilization constrains a HERC from having more than ten Ethernet links per BM, or fourteen Rocket IO links. Counter intuitively, if a HERC with more than fourteen Rocket IO links could be possible of being implemented, its power consumption would be smaller than the power consumed by an equivalent HERC made utilizing Ethernet links. Additionally, except for BUFG resources, this HERC would not consume as many FPGA resources as its Ethernet based equivalent. This conclusion is especially important for future versions of the HERC, where FPGAs with more resources will be utilized.

HERCs with a large number of Rocket IO communication links can be used in large space vehicles, featuring high capacity power distribution systems. In smaller space vehicles, they can be used for applications where processing would only occur in short periods of time. An example of such applications is radar processing, where its high power consumption constrains it to be used only during periodic bursts. The high power consumption of HERCs with Ethernet or RocketIO communication links can be ameliorated by instead using links based on LVDS or Hypertransport pairs. Although serial data transfers up to 3Gbps have been measured in short length LVDS lines, speeds of 600 to 800 Mbps are commonly reached when using this IO standard. The miniaturization of the HERC components will eventually result in short distances between BMs, and thus, short communication links. Therefore, a study of the feasibility of the use of LVDS links with simple serializer/deserializer modules to replace the HERC communication links is recommended.

5.2 Adaptive Wiring Panel.

5.2.1. Improvements of the Routing Algorithms.

The current algorithm is based on a heuristic that uses the shortest path algorithm to get the solution for every connection pair. The order of the connections is first selected arbitrarily. If connections cannot be established, the order of the connections is shuffled in the hope that a solution will be found. This is continuously performed until the connections are established or the permutations are exhausted.

However, it turns out that the algorithm computational time increases exponentially with the number of connection pairs. Moreover, it does not carry out an exhaustive search of the solution space, since each connection pair uses the shortest path (a non-shortest path could lead to more efficient solutions for the next connection pairs).

Now that we have the AWP 2nd Prototype operating, it opens up the possibility of investigating routing algorithms (e.g., Steiner forest) that can provide nearly exhaustive searches of the solution spaces as well as realistic computational times.

5.2.2. Set of Examples with More Detailed Circuits.

The Adaptive Wiring Panel (AWP) has been demonstrated to work with our examples that include resistors, LEDs, and a battery. The next step is to develop a set of examples that include more varied components: DC motors, speakers, solar cells, current meter, alarm IC, etc. The set of examples will contribute to the development of the AWP User Guide.

5.2.3. Integration of the Improved Gesture Interface with the AWP Software.

At the current moment, the gesture interface is used to specify circuit connections for the AWP. It has been shown to work well under certain conditions. The gesture interface is able to recognize circuit connections. However, a more robust interface needs to be developed. In addition, the robust gesture interface software needs to be integrated with the AWP software.

5.2.4. Redesign of the Adaptive Wiring Panel (AWP).

Here, we list a description of enhanced features the next AWP Prototype should have:

- Distributed management approach: A new generation of the AWP that does not require an external master management unit for routing purposes will be developed. This approach will allow local routing decision to solve global routing problems. This implies providing each cell with a processing element (in addition to FPGA fabric) with tight constraints in terms of power consumption and size in order to support the goals of integration and low cost. Several alternatives of hard-core and soft-core processors will be surveyed to determine the best possible fit for this effort.
- Miniaturization: We will assess the feasibility of the integration of the control logic inside the FPGA, the processing element, and the array of solid-state relays (we will be looking for latching relays) in a single die. This miniaturization and integration effort can dramatically reduce the size of the cell and its power consumption. It will also make more space to include all type of connections in the cell, such as optical and RF.
- Incorporation of signals of different nature: This will extend the original AWP concept to a more universal solution, where the switch fabric includes power, optical, and RF signals. This will allow the network to perform in different scenarios and might potentially provide a universal solution for interconnection in aerospace systems.
- Robustness: The Graphic User Interface (GUI) will perform basic testing prior to make the actual circuit connections in order to avoid connections that can damage the circuitry. For instance, user-specified connections that can potentially short battery terminals will be detected and not allowed. The SPICE format of the circuit allows us to perform SPICE simulation to detect damaging connections. In addition, components will be allowed to notify the system of possible failures detected through self-testing. In addition, we will consider the use of low-power transistors for routing analog and digital signals, and the design of low-power relays for routing power.
- Wireless circuit specification for the AWP: A new wireless communication system will eliminate the cable connections to the computer. Then, the finger-movements software routine will be integrated to the monitoring system for specifying circuit connections wirelessly.

REFERENCES

1. Parra, J.E., A reconfigurable multiprocessor architecture for space missions: The AFRL-UNM HERC, PhD Dissertation, The University of New Mexico, Albuquerque, NM, USA, May 2008.
2. Murray, V., Llamocca, D., Lyke, J.C., Avery, K., Jiang, Y., and Pattichis, M., Cell-based architecture for adaptive wiring panels: A first approach, in Reinventing Space, May 2011.
3. Murray, V., Feucht, G.A., Lyke, J., Pattichis, M., and Plusquellic, J., Cell-based architecture for reconfigurable wiring manifolds, in AIAA Infotech@Aerospace, 2010.
4. DeHon, A., Huang, R., and Wawrzynek, J., Hardware-assisted fast routing, in Field-Programmable Custom Computing Machines, 2002.
5. Lyke, J., Wilson, W., and Contino, P., MEMS-based reconfigurable manifold, in NASA MAPLD Conference, 2005.
6. Lyke, J., Wilson, W., and Broyls, R., Adaptive manifold, October 2002.
7. Thompson, S., and Mycroft, I., Self-healing reconfigurable manifolds, in Proc. DCC'06: Designing Correct Circuits, 25-26 March 2006.
8. Nagel, L. W, and Pederson, D.O., "SPICE (Simulation Program with Integrated Circuit Emphasis)", Memorandum No. ERL-M382, University of California, Berkeley, Apr. 1973.
9. The I^2C-bus specification, January 2000.
10. Levitin, Introduction to The Design & Analysis of Algorithms, Addison Wesley, 2003.
11. DeHon, A., Reconfigurable architectures for general-purpose computing, Tech. Rep., AI Technical Report 1586, MIT Artificial Intelligence Laboratory, October 1996.

Appendix A: AWP Software Reference Manual

A.1. Description of the command-line AWP software routine.

A.1.1. Basic concepts: cell array, graph, module array.

We begin with the basic concepts that will be used for describing the AWP software routine:

- *Cell Unit*: It is the smallest unit of the AWP. It consists of an FPGA that takes care of I^2C communications with the computer and its cell neighbors. It also controls the relays.
- *Grid*: It is an array of Cells whose arrangement is mechanically modifiable by the user. In the software routine, it is represented by the structure M.
- *Graph*: It is the representation of the wiring mesh. Each Cell Unit contains the vertices Z1-Z3, Y1-Y12, X1-X16, and U1-U4. In addition, there are 71 relays (considered edges) to connect the vertices. The combination of all the vertices and edges for all cell units makes up the graph. In the software, it is represented by the structure G.
- *Module Array*: It is array of modules (like those in Fig. 33). The modules are attached on top of the cell unit. In software, the module array is represented by Module_List.
- *Cell-level circuit*: It is the set of pairs connected (represented by structure Circ) or to be connected (represented by structure C) at the cell-level, i.e. each pair is represented by <node_name>-<cell ID>.
- *Module-level circuit*: It is the set of pairs connected (represented by the structure Module_List->circuitry) at the module-level, i.e. each pair is represented by <module_node_name>.

Figure 23 shows an example.

A.1.2. Command-line Interface.

A command-line interface has been developed in order to manage the AWP 2nd Prototype shown in Figure 33. In what follows, we provide a tutorial of the available commands.

We initiate the interface using and generate the hardware grid using:

a) mytest_i2c –interface

AWP>>> generate_grid → It creates a connected grid of Cell Units, based on the Cells that were detected. Figure 4(b) shows an example where 6x8 cells were detected.

AWP>>> connect A B → It connects two nodes. It requires the grid to be previously generated. Nodes are provided in the following format: <node_name>-<cell ID>

Example: connect y1-03 y2-1E → Connects node y1 of cell 03 with node y2 of cell 1E

AWP>>> unconnect circuit

Disconnects the current closed relays and regenerates the grid.

AWP>>> unconnect_all

Disconnects all relays and regenerates the grid. This command is useful after the program execution stops due to programming issues. When the interface is restarted, it does not know which relays were left closed.

AWP>>> print circuit → Prints set of connected pairs.

AWP>>> print grid → Prints current grid.

AWP>>> print graph → Prints current graph: each node with its neighbors

AWP>>> exit → Exits interface

AWP>>> modules → Allows the issuing of commands at the module level, i.e. we only set connections on the module pins, or read the module array.

AWP (module level) >>> read modules

For each module, it reads the each module size, its orientation, its components, and the cell to which the module is attached to. Then, it maps the module pins to the corresponding cell pins.

AWP (module level) >>> set_circuit

It allows the specification of a circuit at the module level. A circuit is defined by a set of 'connection pairs'. Each connection pair connects a pin of a component with a pin of another component (or the same component). The format a connection pair is given by:

Component A		Component B	
Name	Module Pin name	Name	Module Pin name

Example: Enter pair >>> R1 y1 R2 z3

Enter pair >>> C1 y1 R1 y2

Enter pair >>> exit

AWP (module level) >>> connect once

This instruction tries to make the list of connections. After making the connections, it gives control back to the user to execute other instructions.

AWP (module level) >>> connect loop

This instruction tries to make the list of connections. After making the connections, it enters into an infinite loop (which can be excited by the user at any time). The program routinely reads the cell grid and module array looking for changes. If there are changes and if the components in the circuit are still in place, it will re-route the paths to connect the circuit.

AWP (module level) >>> print_circuit → Prints circuit (at module level)

AWP (module level) >>> unconnect_circuit → Disconnects current circuit

AWP (module level) >>> exit → Exit module level

An alternative user interface for the AWP can be initiated using:

b) mytest_i2c –interface_2

AWP>>> go → It enters a loop that repeteadly reads and displays the cell grid and the module array. The user can set the circuit at every iteration of the loop. The user can also exit the loop.

When the message 'Press ENTER to set circuit' is displayed, the user can set the circuit by pressing ENTER. A circuit is defined as a collection of connection pairs. The interface allows the user to set several different circuits; this allows for an orderly setup. The user is provided with three choices:

- new: It sets a new circuit
- add: It adds connections to an existing circuit
- mod: It deletes the connections within an existing circuit and sets new connections on that existing circuit

▪ Format of a connection pair:

Component A		Component B	
Name	Module Pin name	Name	Module Pin name

Example: Enter pair >>> R1 Y5 V1 Z1

Enter pair >>> R1 Y8 L1 Y11

Enter pair >>> L1 Z3 V1 Z2

Enter pair >>> exit

▪ After the circuit is set, the loop continues indefinitely, maintaining the circuits as much as possible when the grid and/or module array change. At any iteration, the user can modify, add, or delete the circuits at will.

▪ This interface is very useful for the linkage with a Graphical Unit Interface (GUI) interface. It repeteadly displays the grid array and modules even when no circuit is set.

▪ This interface differs from mytest_i2c –interface, that is more like a debug interface, in which modifying the circuit requires exiting the loop, and read both the grid and module again.

AWP>>> exit: Exits interface

A.1.3. Set of individual commands for AWP control

The following commands take care of writing/reading the information contained in the AT24C08B I^2C PROM memory included in each module. These commands should be used when the user requires the modification of data inside the module. The information consists of: module type and Spice-format list of components inside the module. In addition, it requires direct connection of the I^2C pins of the VGA connector of the computer with the I^2C pins of the to-be-modified module board.

mytest_i2c –write_AT {# of Spice datasheet}

It writes data provided in a text file to the I^{2}C PROM. No more than 256 characters can be written. The Spice datasheets are provided in a text file named I2Cprom-{number}.txt.

Example of a text file for a module of size 5x10 and 4 components:

```
5x10
R3 V1 V2 330
D3 V8 W3 HLMP-C100
D4 Y11 Y8 HLPM-C100
V2 W1 W2 6v
```

mytest_i2c –read_AT

Reads data from a I^{2}C PROM. This is useful to verify the state of a I^{2}C PROM.

The interfaces described in Section A.1.2 made use of a set of functions that deal with the graph, the cell array (or grid), and the module array. For debugging purposes, some of those functions can also be executed from the command-line, albeit individually. Examples are given below.

mytest_i2c <cell ID> -relays –{off,on}

This opens (off) or closes (on) all relays of the indicated cell (cell ID)

mytest_i2c –relays {off,on}

This opens (off) or closes (on) all relays of every available cell

mytest_i2c -scan

It scans cells from address 2 to 127 (02→7F) and lists whether or not each cell exists.

mytest_i2c <cell ID> -neighbors → Get neighbors' IDs (if any) of a cell defined by 'cell ID'

mytest_i2c –grid → It creates a connected cell array (grid) with all the cells.

mytest_i2c -connect <circuit number>

Given a circuit (set of connections pairs), it connects all the pairs (if possible) by finding the shortest path (Dijkstra's algorithm) between them inside the graph

Notes: <circuit number> : Set of pre-defined circuits (1-6 are allowed)

Example: mytest_i2c -connect 1

Closes the relays in cell 'AB' so that all the pairs in 'circuit 1' are connected

mytest_i2c –m <cell ID> → It reads and displays module information from indicated cell.

mytest_i2c <cell ID> -relay <relay number> -{open, closed, status}

Open, close, or get status of a relay in a cell

Notes: <cell ID> is an hexadecimal number: 02 → 7F

<relay number> is a decimal number: 1 → 71

Examples:

mytest_i2c AB -relay 64 –open → Opens relay 64 from cell 0xAB

mytest_i2c 0C -relay 12 –closed → Closes relay 12 from cell 0x0C

mytest_i2c 10 -relay 32 –status → Returns status of relay 32 of cell 0x10

A.1.4. Commands for I^{2}C communication with the AWP.

The following commands handle the basic I^{2}C communication between a computer and any I^{2}C peripheral. The user can write a byte to a given I^{2}C peripheral or read a byte from a I^{2}C peripheral. For the AWP, the peripheral can be either the FPGA inside the cell or the I^{2}C PROM inside the module. The commands listed in the previous section rely on these commands to communicate with the I^{2}C peripherals. In addition, they are very useful for debugging purposes.

mydebug_i2c -w <cell ID> <data>

Writes a byte (data) on a cell defined by 'cell ID'

mydebug_i2c -r <cell ID>

Reads a byte from a cell defined by 'cell ID'

Notes: <data> is an hexadecimal number: 00 → FF

<cell ID> is an hexadecimal number: 00 → 7F

cell ID=00 → The command is broadcast to each cell. When reading the routine grabs data from the 1st cell that responds.

cell ID=01 is not allowed since 0x01 means inexistent cell.

Examples: mydebug_i2c -w 02 FA → writes 0xFA on address 0x02

mydebug_i2c -r 03 → reads a byte from address 0x03

mydebug_i2c –s → Runs a scan of cells (02→7F) and lists the cells that exist.

A.1.5. Code structure and libraries description

The following is the file structure of the AWP command-line software. The ‘Makefile’ file takes care of the configuration for the ‘gcc’ compiler. Two executables are obtained: ‘mydebug_i2c’, and ‘mytest_i2c’.

```
mydebug_i2c.c
        i2c.h
        i2c-linux.c
        my_i2c_commands.h
mytest_i2c.c
```

i2c.h

i2c-linux.c

my_i2c_commands.h

my_i2c_macros.h

user_interface_functions.h

module_level_functions.h

grid_creation_functions.h

shortest_path_functions.h

priority_queue_functions.h

The file i2c-linux.c contains the low-level functions to communicate with the I^2C port of the computer. The i2c.h file is the header that lists those low-level functions. The other .h files are explained below.

my_i2c_commands : It includes the basic functions for I^2C communication.

int debug_i2c(unsigned char address, unsigned char *data, char op)

Performs a I^2C read or write from/to address based on op ('w' or 'r').

int TypeI_fun(unsigned char cellID, unsigned char command,

unsigned char relay, int oc)

It writes a command and data (concatenation of relay number and close/open operation) to the I^2C bus. No data retrieved.

int TypeII_fun(unsigned char cellID, unsigned char command,

unsigned char *o_data)

It writes a command to the I^2C bus. Then, it receives a byte from I^2C bus.

int TypeIII_fun(unsigned char cellID, unsigned char command,

unsigned char **o_data)

It writes a command to the I^2C bus. Then, it receives a byte specifying the number of bytes 'n' to receive. Finally, it receives 'n' bytes.

int TypeIV_fun(unsigned char cellID, unsigned char command,
unsigned char i_data, unsigned char *o_data)

It writes command and data to the I^2C bus. Then, it gets a byte from I^2C bus.

int TypeV_fun(unsigned char word_address, unsigned char *data, char op)

It reads data from the Atmel two-wire serial EEPROM AT24C08B. In other words, it sends a command to the FPGA inside a cell to read data from the module attached to it and forwards it to the computer. This command is very different from mytest_i2c –read_AT, which is a stand-alone function that requires direct connection of the VGA port of the computer and the module pins.

my_i2c_macros: It includes advanced functions for management of the AWP. It makes use of the functions defined in all the other .h files.

int scan_neighbors(unsigned char address)

It prints the IDs of the neighbors (if any) of the cell identified with 'address'.

int scan_cells()

It prints the list of all cells (from 2 to 127) indicating whether or not each one exists.

int connect_graph(int circuit_number)

It attempts to connect the list of pairs given by the circuit defined by 'circuit number'.

int user_interface()

It calls the user interface (cell level) described in Section A.1.2.a

int user_interface_v2()

It calls the user interface (cell level) described in Section A.1.2.b

user_interface_functions: It includes high-level functions that perform user commands.

int ui_generate_grid(Graph *G, Grid *M)

It builds the grid (array of cells)and creates the graph based on the grid.

int ui_unconnect(char* flag, Grid *M)

It disconnects relays. flag='circuit' → disconnects current closed relays. flag='all' → disconnects all relays.

int ui_connect (Graph *G, Grid *M, Circuit *C)

It connects a list of pairs (provided by structure C) using the shortest-path algorithm.

module_level_functions: It includes high-level functions that deal with management of the module array.

int module_print_circuit (Module_array *Module_List, Circuit* Circ)

It prints the module-level circuit (provided by Circ)

int module_define_circuit(Module_array *Module_List, Circuit *C)

It accepts a list of pairs at module-level (user-input) and returns a list of pairs at the cell-level, which is provided in C.

void module_print(Module_array *Module_List)

It prints a list of modules with all their information int module_map(Grid *M, Module_array *Module_List)

It converts a set of module-level pairs into cell-level pairs. The information is returned into Module_List.

int module_read(Grid *M, Module_array *Module_List)

It reads the available modules and retrieves the information into Module_List.

grid_creation_functions: It includes advanced functions that deal with the cell array (or grid)

int grid_shuffle_circuit(Circuit *C, Permut *P)

It permutes the list of connected pairs (provided in structure C) according to the permutation provided by structure P.

two_data grid_relay_get_number(char *A, char* B, Grid *M)

It gets the index of the relay that connects pins A and B.

void grid_relay_initialize (Grid *M)

It builds a table of all possible pin pairs so that a relay can be identified later.

int grid_create_graph(Graph* G, Grid* M)

It builds the undirected graph that represents the Adaptive Wiring Panel wiring mesh. The graph can be built only after the cell arrangement is known.

int grid_get_neighborhood(Grid *M)

It reads every available cell along with its neighbors and stores them in a structure that represents the array of cells.

int grid_build(Grid *M)

It creates the grid (or array of cells) based on the information provided by ‘grid_get_neighborhood’. The grid is defined by the list of cells as well as the arrangement which is obtained by this function.

shortest_path_functions: It includes the functions that add and delete nodes from a graph, get vertex value, and get the shortest path solution for a connection pair.

int graph_get_vertex_value (Graph *G, char* pin_name)

It gets the node number at which 'pin_name' is located in the current graph.

int graph_my_dijkstra(Graph* G, int source, int**T, int* d)

Solves the one-to-all shortest path problem on graph G using Dijkstra's algorithm. It returns the edges and the key values of each node.

void graph_print_results(int source, int edges, int nvertices, int **T, int *d, char** name_vertices)

It prints the one-to-all shortest path solution provided by ‘graph_my_dijkstra’.

void graph_add_node(Graph* G, char *node_name, int n_neighbors, int* neighbors, int *weights)

It adds a node to graph G (and the outcoming and incoming paths from/to the node)

void graph_delete_node(Graph* G, int node)

It deletes a node from the current graph (and the outcoming and incoming paths from/to the node).

priority_queue_functions: holds the functions that implement a minimum priority queue. These functions are used by 'graph_my_dijkstra' defined in the shortest_path_functions module.

int parent (int i)

Gets the parent of a node.

void min_heapify(two_data* heap, int *heap_size, int i)

Let the value at heap[i] float down in the min-heap so that the subtree rooted at 'i' becomes a min-heap.

void build_min_heap(two_data *heap, int *heap_size)

Given a heap with data, builds a min-heap.

int heap_extract_min (two_data *heap, int *heap_size)

The node with the minimum key is extracted from the heap.

void heap_decrease_key (two_data *heap, int i, int key)

Decrease the key of a node at position 'i'.

A.2. Description of the Graphic Unit Interface (GUI) for the AWP project.

The GUI system for AWP is a layer added to the command-line interface to enhance the user experience and to help visualize the working mechanism inside the system. It currently consists of one *mouse-controlled interface* and one *camera capture interface*. The mouse-controlled interface allows for complete control of the AWP. The camera interface that allows us to control the AWP via finger movement still experiences some problems, and as such, it has not been integrated with the Graphic Unit Interface (GUI).

The *mouse-controlled interface* is shown in Figure A-1. The menu bar allows for some basic relay operation, help, etc. The Control Area includes a set of buttons that let the user modify the view and the operation type. The Display Area shows specific information about the AWP system.

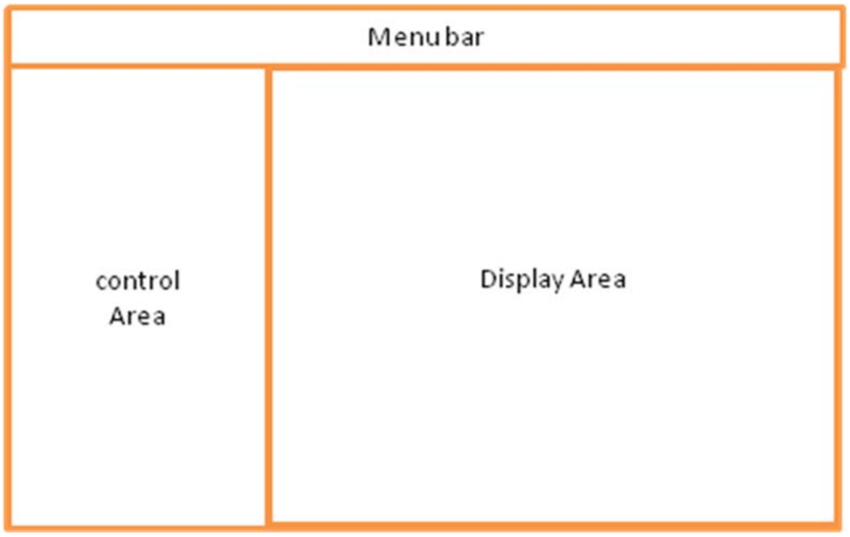
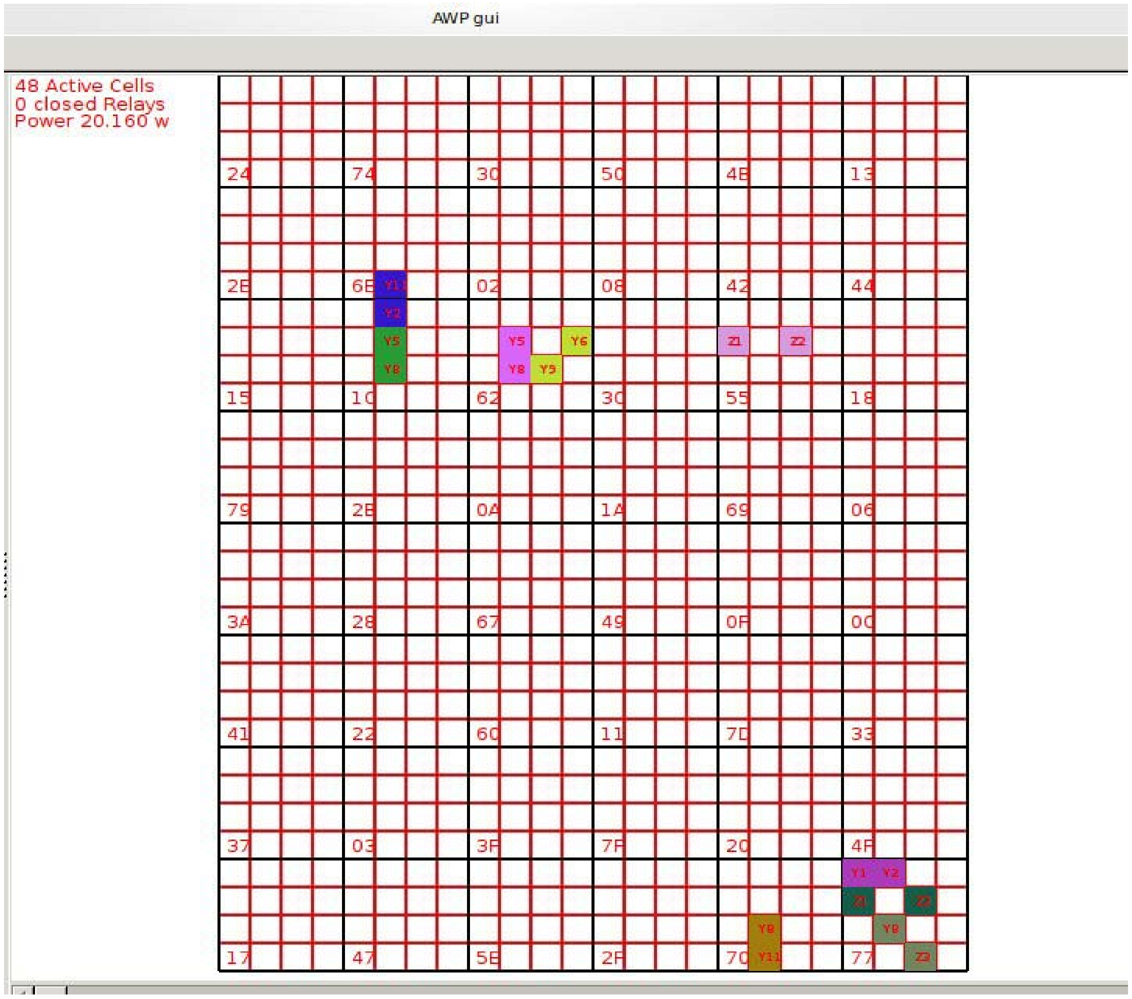


Figure A-1. Layout of the mouse-controlled interface.

The Display Area can show four (4) specific views, called ‘layers’:

- **Cell Array Layer**: It shows the cells that are connected in the AWP system. Each cell is displayed as rectangle consisted of 4x4 small rectangles. Inside each cell, the lower-left rectangle side rectangle shows the cell ID. The other rectangles shown are the 15 pins on each cell. If a component exists in the AWP, the user can see its pins colored with same color and shown on the cell array. A snapshot of this layer is shown in Figure A-2; here, 6x8 cells are interconnected. The idea is that the cell array is shown in real time, e.g. if a cell is removed, its corresponding rectangle will disappear from the view.
- **Module Layer**: It shows the positions and orientations of the module boards plugged into the cell array. Two kinds of module boards (5x5 and 5x10) and four rotations (0º, 90º, 180º, and 270º) are supported. The module position in this layer matches the cell position in cell array layer. Each module board shows an ID, this is the ID of the cell the module is plugged in. An example of this layer is shown in Figure A-3. Four module boards shown, two of them are 5x5 boards while two others are 5x10 boards.
- **Circuit Layer**: This layer shows the circuit components in the module boards. Each component appears with its name on its side. When this layer is in connection mode, the user can make connections by clicking on the terminals of each component (there is a small rectangle for each terminal of a component). When a terminal is clicked, the small rectangle is highlighted. A snapshot of this layer is shown in Figure A-4.
- **Link Layer**: This layer shows the cells and the relays that are currently closed. If there are relays closed inside a cell, the cell turns green. By double-clicking the specific cell, the user can see the cell in detail with the condition of the relays (closed/open). An example is shown in Figures 37, 38, and 39 . Figure 37 shows an example circuit array connected in the system. Figure 38 shows the cells in which their relays are closed. Figure 39 shows the relays inside cell 62, the highlighted rectangle denotes a closed relay.



```
48 Active Cells
0 closed Relays
Power 20.160 w
```

Figure A-2. Cell Array Layer, 9 components are plugged on the AWP.

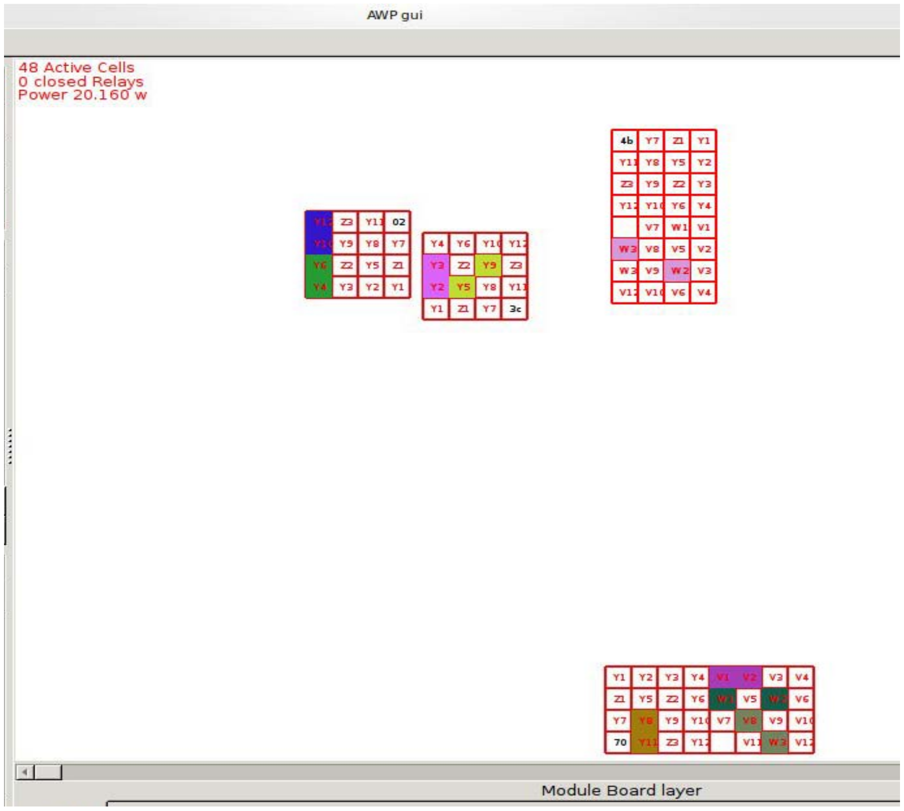


Figure A-3. Module Array Layer.

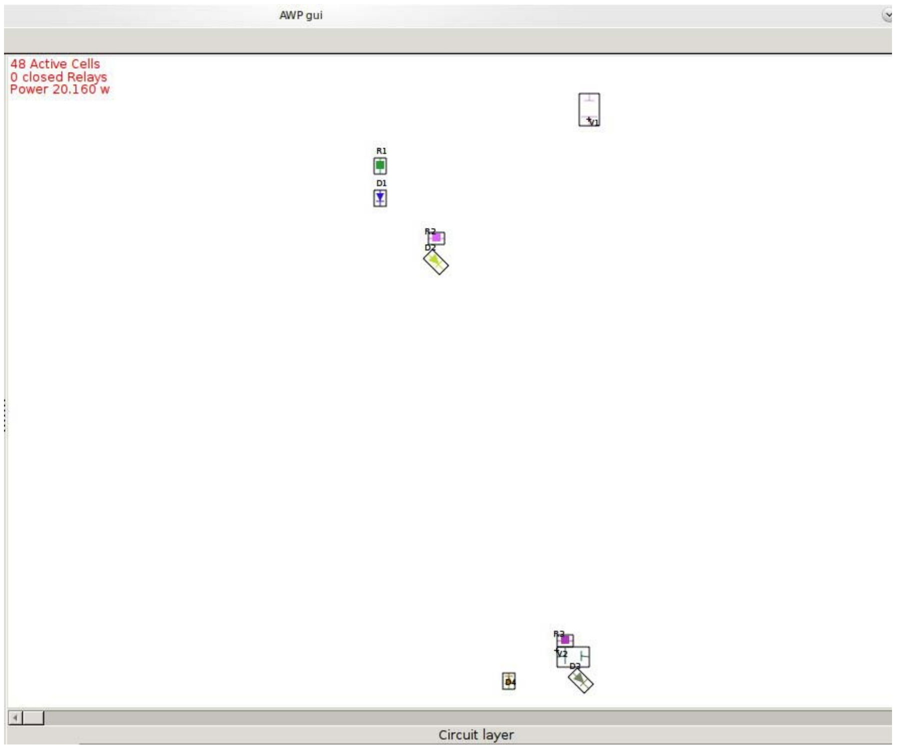
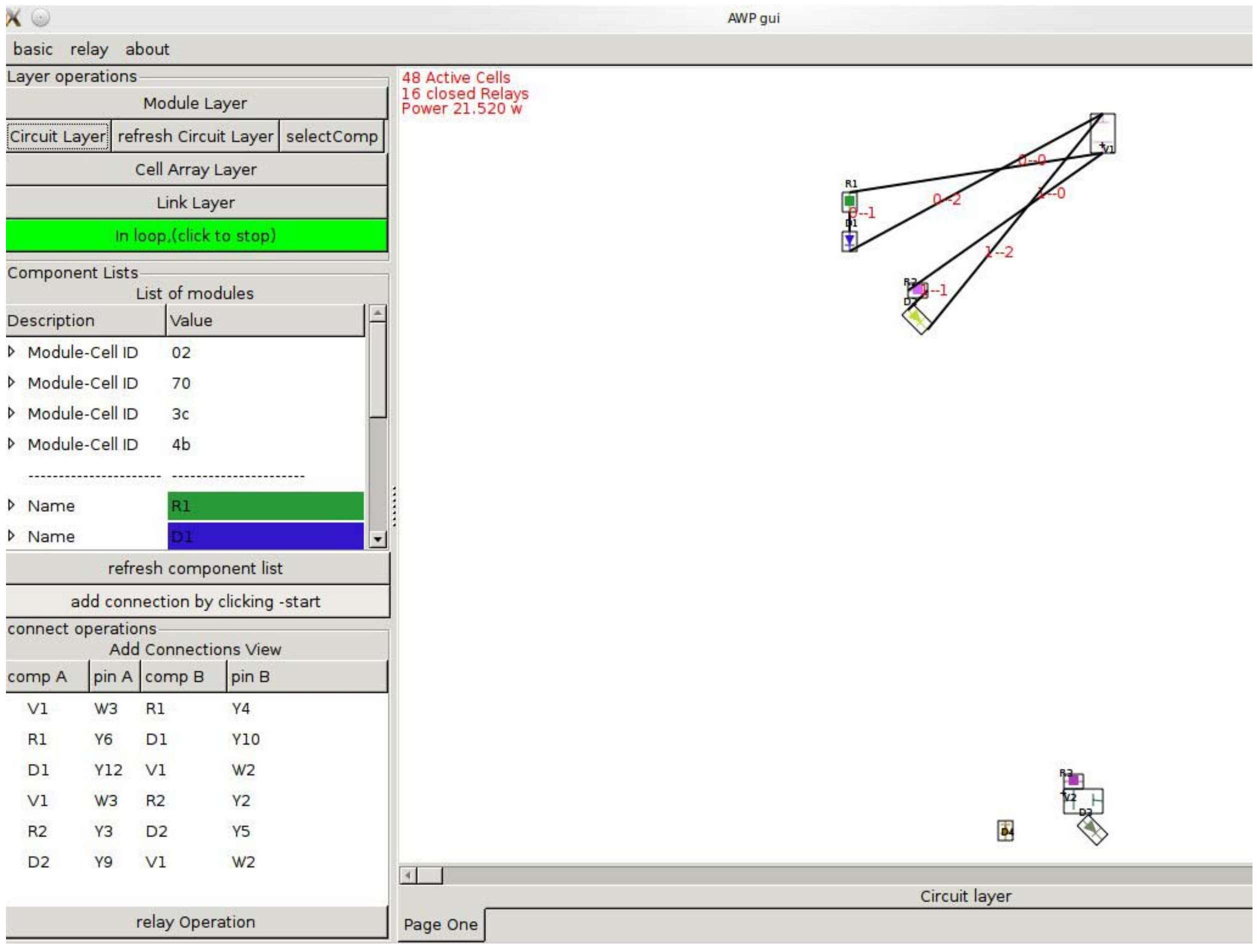


Figure A-4. Circuit Layer.

The control area is shown in Figure A-5. There are 3 separate framed areas in this area.

- **Layer operations**: This area contains 4 buttons to change from layer to layer, each of them stands for one layer. Specially, for circuit layer, there are also buttons to move the components around and initialize them according to the user specifications. There is also the "start loop" button which controls the communication loop. It will turn green when the loop is running, and it will turn grey if the loop is stopped or paused.

- **Component Lists**: This area contains a list that shows the modules and components in the AWP. There is a button to manually refresh the list. Also, there is a button to add connections in the circuit layer. Once the 'add connection' button is pressed, it will change color to red, the user waits until the 'add connection' notice dialog box shows up. Once the drawing is done, the user presses the same button which now is labeled as 'finished connection'. At this moment, the AWP will get the user input and try to establish the connections.
- **Connect operations**: This area contains a list shows the connection made. It also contains a button to make relay operations. Once a the relay button is pressed, a dialog will show up and ask for 4 operations: i) open all relays, ii) close all relays, 3) open all relays in a cell, and iv) close all relays in a cell.



48 Active Cells
16 closed Relays
Power 21.520 w

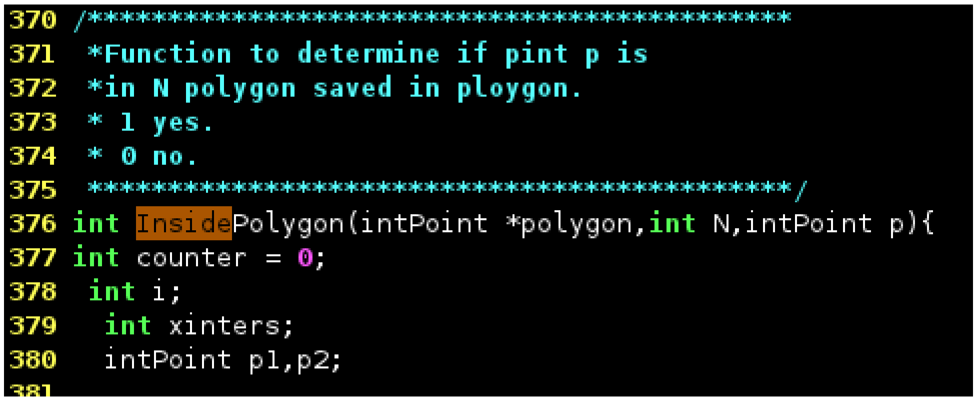
Description	Value
Module-Cell ID	02
Module-Cell ID	70
Module-Cell ID	3c
Module-Cell ID	4b
---------------------	---------------------
Name	R1
Name	D1

comp A	pin A	comp B	pin B
V1	W3	R1	Y4
R1	Y6	D1	Y10
D1	Y12	V1	W2
V1	W3	R2	Y2
R2	Y3	D2	Y5
D2	Y9	V1	W2

Figure A-5. Control Area.

Program structure

The mouse controlled GUI is just a GTK wrapper for the command line system. The current version is V4.4. All the functions implemented are in '*gui_command.h*' and '*gui_test_i2c.c*'. In '*gui_test_i2c.h*', the main function is implemented and some global variables are declared. The gtk_main loop is executed in the main function. In '*gui_command.h*', all accessory functions about the drawing and responding are included. The structure of the whole code is such that the *gtk_main()* loop takes care of the user response when an event occurs (button click, loop request to redraw area, etc.). Then, the corresponding registered code will be called to respond to it. An example function is shown in Figure A-6, which is used to determine if a point 'p' is inside the polygon or not.



```
/********************************************
 *Function to determine if pint p is
 *in N polygon saved in ploygon.
 * 1 yes.
 * 0 no.
 ********************************************/
int InsidePolygon(intPoint *polygon,int N,intPoint p){
int counter = 0;
 int i;
  int xinters;
  intPoint p1,p2;
```

Figure A-6. Coding Style.

Appendix B: AWP Hardware Reference Manual

B.1. Block Diagram of the Hardware Design in the FPGA.

We present in Figure 1 the block diagram of the hardware design implemented in the FPGA using VHDL. The block diagram shows the dependency of all the VHDL blocks included in the design. Each block is explained in the next subsections.

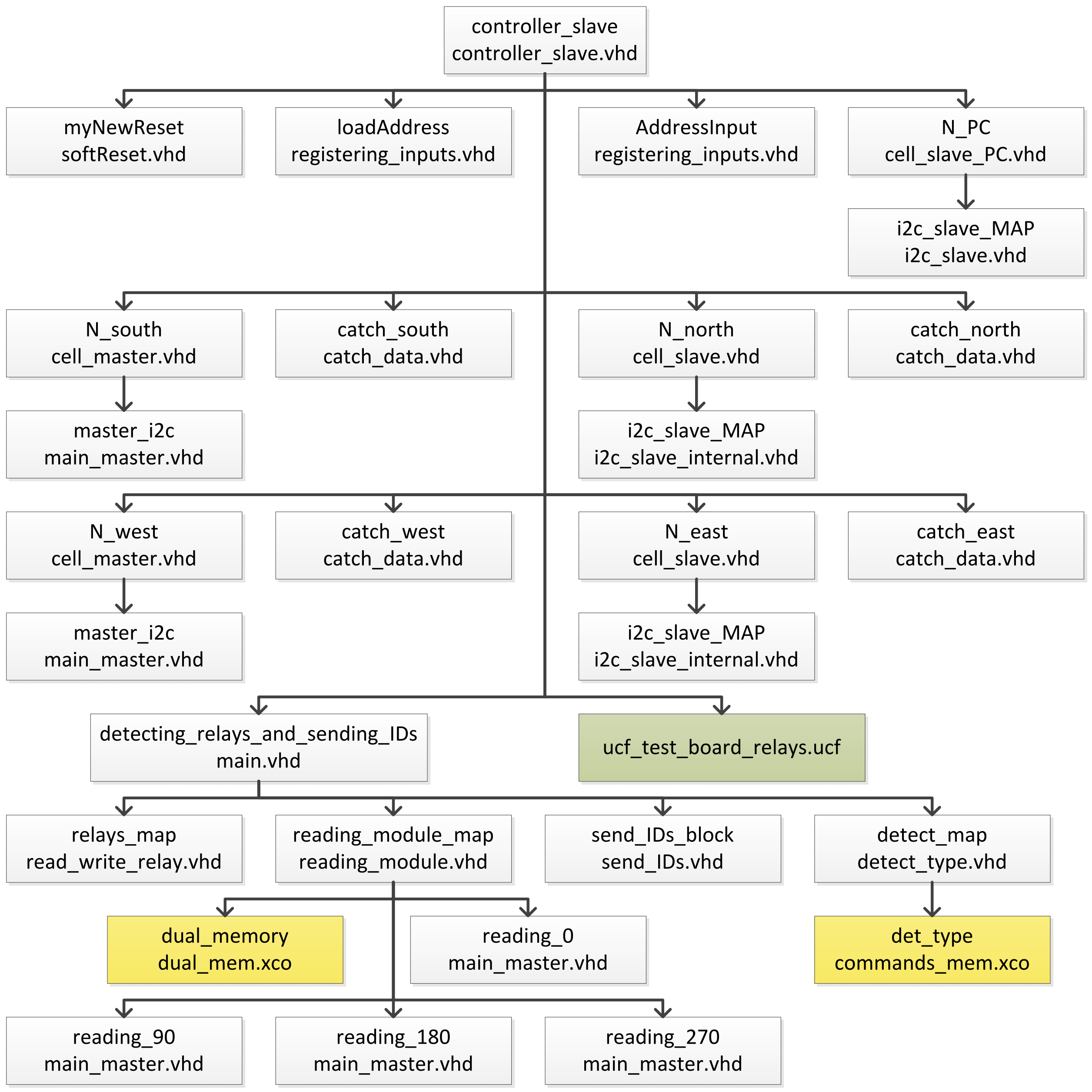


Figure B-1. Dependency diagram of the hardware blocks implemented with VHDL.

B.2 controller slave

This is the main VHDL file. This module connects every single description file of the AWP project. The file used is "controller_slave.vhd".

The entity of the file is described as:

```
entity controller_slave is
       Generic (Nbits_data : natural := 8;
                    (Number of bits for the data in the I2C connection.)
                Nbits_address : natural := 7;
                    (Number of bits for the address in the I2C connection.)
                N_relays : natural := 128);
                    (Maximum number of relays per cell.)
       Port ( resetn : in  STD_LOGIC;
                    (External reset in negative logic.)
              clock: in std_logic;
                     (External clock oscillator @ 50MHz.)
              addressn : in  STD_LOGIC_VECTOR(7 downto 0);
                     (Address vector to define the ID of the cell.)
              sda_pc : inout  STD_LOGIC;
                     (SDA signal to the main PC.)
              scl_pc : in  STD_LOGIC;
                     (SCL signal to the main PC.)
              sda_North,sda_East,sda_South,sda_West : inout  STD_LOGIC;
                     (SDA signal to the 4 cell neighbors.)
              scl_North : in  STD_LOGIC;
                     (SCL signal to the north neighbor.)
              scl_East : in  STD_LOGIC;
                     (SCL signal to the east neighbor.)
              scl_South : out  STD_LOGIC;
                     (SCL signal to the south neighbor.)
              scl_West : out  STD_LOGIC;
                     (SCL signal to the west neighbor.)
              relaysp : out  STD_LOGIC_VECTOR (70 downto 0);
                     (Signals to control each external relay.)
              scl_module_0 : out  STD_LOGIC;
                     (SCL signal to detect the component at 0º.)
              sda_module_0 : inout  STD_LOGIC;
                     (SDA signal to detect the component at 0º.)
              scl_module_90 : out  STD_LOGIC;
                     (SCL signal to detect the component at 90º.)
              sda_module_90 : inout  STD_LOGIC;
                     (SDA signal to detect the component at 90º.)
               scl_module_180 : out  STD_LOGIC;
                    (SCL signal to detect the component at 180º.)
               sda_module_180 : inout  STD_LOGIC;
                    (SDA signal to detect the component at 180º.)
               scl_module_270 : out  STD_LOGIC;
                    (SCL signal to detect the component at 270º.)
               sda_module_270 : inout  STD_LOGIC
                    (SDA signal to detect the component at 270º.)
                    );
        end controller_slave;
```

B.3 myNewReset

This block uses the file "softReset.vhd". The goal of this block is to produce an automatically internal reset of all the blocks in the AWP system once they are powered by first time.

This block was added after the finding of the internal problem of the no synchronization of all the blocks when they were powered.

Currently, the block is set to wait for 1 second before producing the 1-second internal reset. This internal reset is produced only once.

The entity of the file is described as:

```
entity softReset is
      Generic (
         waitingLow     :     integer := 1; -- Time in SECONDS to wait
               (Time in seconds to wait before producing the internal reset.)
         longPulse      :     integer := 1-- Time in SECONDS to 'push' reset
               (Time in seconds to 'press' the internal reset.)
            );
      Port ( reset : in  STD_LOGIC;
                (External reset in positive logic.)
           clk50M : in  STD_LOGIC;
                (External clock oscillator @ 50MHz.)
           newReset : out  STD_LOGIC);
                (New internal reset in positive logic to be produced.)
```

```
end softReset;
```

B.4 loadAddress

This block uses the file "registering_inputs.vhd". This block is an internal register to catch the signal lo load the ID designed to the cell.

The entity of the file is described as:

```
entity registering_inputs is
    Generic ( Nbits : natural := 1
                (Number of bits desired in the registered buffer.)
                        );
    Port ( reset : in  STD_LOGIC;
                (External reset in positive logic.)
           clk : in  STD_LOGIC;
                 (External clock oscillator @ 50MHz.)
           A : in  STD_LOGIC_VECTOR (Nbits-1 downto 0);
                 (Input signal.)
           B : out  STD_LOGIC_VECTOR (Nbits-1 downto 0));
                 (Output signal.)
end registering_inputs;
```

B.5 AddressInput

This block uses the file "registering_inputs.vhd". This block is an internal registered bus to catch the ID designed to the cell.

B.6 N_PC

This block uses the file "cell_slave_PC.vhd". This block is in charge of the communication with the external PC using the I^{2}C protocol. The timing and signal types of this block were adapted to the signals produced by the I/O from the laptop using Linux. This block contains an internal state machine to control the communication with the PC.

This block controls the block "i2c_slave_MAP", which uses the file "i2C_slave.vhd". This VHDL file is set up to work as a slave in the I^{2}C protocol.

The entity of the file "cell_slave_PC.vhd" is described as:

```
entity cell_slave_PC is
      generic (
          Nbits_data : natural := 8;
                 (Number of bits for the data in the I2C connection.)
          Nbits_address : natural := 7);
                 (Number of bits for the address in the I2C connection.)
  Port ( clk50M : in  STD_LOGIC;
                 (External clock oscillator @ 50MHz.)
         reset : in  STD_LOGIC;
                 (New internal reset in positive logic.)
         load_address : in  STD_LOGIC;
                 (Internal signal to load the address (ID of the cell).)
         address : in
            STD_LOGIC_VECTOR (Nbits_address-1 downto 0) := (others => '0');
                 (Address (ID of the cell).)
         load_data : in  STD_LOGIC;
                 (Internal signal to load the data to be sent.)
         data_in : in  STD_LOGIC_VECTOR (Nbits_data-1 downto 0) := x"01";
                 (Data to be sent.)
         scl : in  STD_LOGIC;
                 (SCL signal to the main PC.)
         data_ready : out std_logic;
                 (Data read ready in the register.)
         data_out : out  STD_LOGIC_VECTOR (Nbits_data-1 downto 0) := x"01";
                 (Data read.)
         sda : inout  STD_LOGIC;
                 (SDA signal to the main PC.)
         busy : out STD_LOGIC);
                 (Signal to indicate that the block is busy: a transfer is
                  being processed.)
end cell_slave_PC;
```

The entity of the file " i2C_slave.vhd" is described as:

```
entity i2c_slave is
      generic (
            Nbits_data : natural := 8;
                 (Number of bits for the data in the I2C connection.)
            Nbits_address : natural := 7);
                 (Number of bits for the address in the I2C connection.)
    port ( reset : in  STD_LOGIC;
                 (New internal reset in positive logic.)
           clk50M : in  STD_LOGIC;
                 (External clock oscillator @ 50MHz.)
           scl : in  STD_LOGIC := '1';
                 (SCL signal for the I2C protocol.)
           sda : inout std_logic;
                 (SDA signal for the I2C protocol.)
           load_address : in  STD_LOGIC;
                 (Internal signal to load the address to be sent.)
           address : in  STD_LOGIC_VECTOR (Nbits_address-1 downto 0);
                 (Address to be used.)
           load_data : in  STD_LOGIC;
                 (Internal signal to load the data to be sent.)
           data_in : in  STD_LOGIC_VECTOR (Nbits_data-1 downto 0);
                 (Data to be sent.)
           data_out : out  STD_LOGIC_VECTOR (Nbits_data-1 downto 0);
                 (Data read.)
           data_read : out  STD_LOGIC;
                 (Data read ready in the register.)
           busy : out  STD_LOGIC);
                 (Signal to indicate that the block is busy: a transfer is
                 being processed.)
end i2c_slave;
```

B.7 N_south

This block uses the file "cell_master.vhd". This block is in charge of controlling the communication with the north neighbor. This block contains an internal state machine to control the communication with the other neighbor. The file uses the block "master_i2c", which depends on the file "main_master.vhd". The latter VHDL file is a master block to control I^{2}C communications.

The entity of the file "cell_master.vhd" is described as:

```
entity cell_master is
       Generic (
            Nbits_data : natural := 8;
                (Number of bits for the data in the I2C connection.)
            Nbits_address : natural := 7);
```

```
                (Number of bits for the address in the I2C connection.)
    Port ( reset : in  STD_LOGIC;
                (New internal reset in positive logic.)
           clk50M : in  STD_LOGIC;
                (External clock oscillator @ 50MHz.)
           enable : in  STD_LOGIC := '1';
                (Enable the block.)
           sda : inout  STD_LOGIC := '1';
                (SDA signal.)
           scl : out  STD_LOGIC;
                (SCL signal.)
           data_ready : out  STD_LOGIC;
                (Data read ready in the register.)
           data_out : out
              STD_LOGIC_VECTOR(Nbits_data-1 downto 0) := (others => '0');
                (Data read.)
           my_ID : in std_logic_vector(Nbits_data-1 downto 0) := "01101001"
                (ID to be set up.)
                    );
end cell_master;
```

The entity of the file " main_master.vhd" is described as:

```
entity main_master is
      generic (
            Nbits_data : natural := 8;
                 (Number of bits for the data in the I2C connection.)
            Nbits_address : natural := 7);
                 (Number of bits for the address in the I2C connection.)
   Port ( enable_master : in  STD_LOGIC;
                 (Enable signal.)
           readwrite_01 : in  STD_LOGIC;
                 (Signal to select if a Read or Write transfer will be
                  performed.)
           reset : in  STD_LOGIC;
                 (New internal reset in positive logic.)
           clk50M : in  STD_LOGIC;
```

```
                  (External clock oscillator @ 50MHz.)
            scl : out  STD_LOGIC := '1';
                  (SCL signal.)
            sda : inout  STD_LOGIC := '1';
                  (SDA signal.)
            load_address : in  STD_LOGIC;
                  (Internal signal to load the address to be sent.)
            address : in  STD_LOGIC_VECTOR (Nbits_address-1 downto 0);
                  (Address to be used.)
            load_data : in  STD_LOGIC;
                  (Internal signal to load the data to be sent.)
            data_in : in  STD_LOGIC_VECTOR (Nbits_data-1 downto 0);
                  (Data to be sent.)
            data_out : out
                  STD_LOGIC_VECTOR (Nbits_data-1 downto 0) := (others => '0');
                  (Data read.)
            data_read : out  STD_LOGIC := '0';
                  (Data read ready in the register.)
            busy : out  STD_LOGIC := '1');
                  (Signal to indicate that the block is busy: a transfer is
                   being processed.)
end main_master;
```

B.8 catch_south

This block uses the file "catch_data.vhd". This blocks works as register to store the data read by the I^{2}C block described before. The goal is to reduce the noisy info that the block might be reading. If the block doesn't detect a new data in 't' us, then the other module was disconnected and a "0x01" value is produced.

The entity of the file " catch_data.vhd" is described as:

```
entity catch_data is
       Generic (
             cycles_to_wait : integer := 25
                  (Time to wait for a new data. In the example,
                    t = 25 = 250us.)
                      );
    Port ( reset : in  STD_LOGIC;
                  (New internal reset in positive logic.)
```

```
           clk50M : in  STD_LOGIC;
                 (External clock oscillator @ 50MHz.)
           ready : in  STD_LOGIC;
                 (Data from the I2C block is ready.)
           data_in : in  STD_LOGIC_VECTOR (7 downto 0);
                 (Data read.)
           data_out : out  STD_LOGIC_VECTOR (7 downto 0)
                 (Data registered.)
                    );
end catch_data;
```

B.9 N_north

This block uses the file "cell_slave.vhd". The goal of this block is to communicate with the south neighbor. This block controls the internal block "i2c_slave_MAP", which uses the file "i2c_slave_internal.vhd". This I^2C block works as a slave for the I^2C communication.

The entity of the file "cell_slave.vhd" is described as:

```
entity cell_slave is
      generic (
            Nbits_data : natural := 8;
                 (Number of bits for the data in the I2C connection.)
            Nbits_address : natural := 7);
                 (Number of bits for the address in the I2C connection.)
      Port (
           clk50M : in  STD_LOGIC;
                   (External clock oscillator @ 50MHz.)
           reset : in  STD_LOGIC;
                   (New internal reset in positive logic.)
           load_address : in  STD_LOGIC;
                   (Internal signal to load the address to be sent.)
           address : in
              STD_LOGIC_VECTOR (Nbits_address-1 downto 0) := (others => '0');
                   (Address to be used.)
           scl : in  STD_LOGIC;
                   (SCL signal.)
      data_ready : out std_logic;
                   (Data read ready in the register.)
           data_out : out
```

```
                STD_LOGIC_VECTOR (Nbits_data-1 downto 0) := (others => '0');
                    (Data read.)
            sda : inout  STD_LOGIC);
                    (SDA signal.)
end cell_slave;
```

The entity of the file " i2c_slave_internal.vhd" is described as:

```
entity i2c_slave_internal is
      generic (
            Nbits_data : natural := 8;
                  (Number of bits for the data in the I2C connection.)
            Nbits_address : natural := 7);
                  (Number of bits for the address in the I2C connection.)
    port ( reset : in  STD_LOGIC;
                  (New internal reset in positive logic.)
           clk50M : in  STD_LOGIC;
                  (External clock oscillator @ 50MHz.)
           scl : in  STD_LOGIC := '1';
                  (SCL signal.)
      sda : inout std_logic;
                  (SDA signal.)
           load_address : in  STD_LOGIC;
                  (Internal signal to load the address to be sent.)
           address : in  STD_LOGIC_VECTOR (Nbits_address-1 downto 0);
                  (Address to be used.)
           load_data : in  STD_LOGIC;
                  (Internal signal to load the data to be sent.)
           data_in : in  STD_LOGIC_VECTOR (Nbits_data-1 downto 0);
                  (Data to be sent.)
           data_out : out  STD_LOGIC_VECTOR (Nbits_data-1 downto 0);
                  (Data read.)
           data_read : out  STD_LOGIC;
                  (Data read ready in the register.)
           busy : out  STD_LOGIC);
                   (Signal to indicate that the block is busy: a transfer is
                    being processed.)
```

```
end i2c_slave_internal;
```

B.10 catch_north, N_west, catch_west, N_east, catch_east

These blocks are similar to the previous 3 blocks described.

B.11 detecting_relays_and_sending_IDs

This block is the 'brain' of the hardware. It decodes the instructions sent by the external PC and controls the blocks to control the external relays to perform the routing. The main file used is "main.vhd". Each controlled sub-block here will be described in the next subsections.

The entity of the file "main.vhd" is described as:

```
entity main is
       Generic (
            N_relays : natural := 128;
                 (Maximum number of relays per cell.)
            Nbits_data : natural := 8
                 (Number of bits for the data in the I2C connection.)
                         );
      Port ( reset : in  STD_LOGIC;
                 (New internal reset in positive logic.)
           clk50M : in  STD_LOGIC;
                 (External clock oscillator @ 50MHz.)
           busy_PC : in  STD_LOGIC;
                 (I2C from PC is busy.)
           data_ready_PC : in  STD_LOGIC;
                 (Data ready from I2C from PC.)
           data_in : in  STD_LOGIC_VECTOR (7 downto 0);
                  (Data to be transfer.)
           data_out : out  STD_LOGIC_VECTOR (7 downto 0);
                  (Data read.)
           load_data_PC : out  STD_LOGIC;
                  (Load data from I2C from PC.)
            my_ID, north_ID, east_ID, south_ID, west_ID, module_ID : in
                  STD_LOGIC_VECTOR(Nbits_data-1 downto 0);
                  (IDs from each cell, including the 4
                      neighbors and the module.)
            relays : out  STD_LOGIC_VECTOR (N_relays-1 downto 0);
```

```
                     (External relays to be controlled.)
             scl_module_0 : out  STD_LOGIC;
                     (SCL signal to detect the component at 0º.)
            sda_module_0 : inout  STD_LOGIC;
                     (SDA signal to detect the component at 0º.)
            scl_module_90 : out  STD_LOGIC;
                     (SCL signal to detect the component at 90º.)
            sda_module_90 : inout  STD_LOGIC;
                     (SDA signal to detect the component at 90º.)
            scl_module_180 : out  STD_LOGIC;
                     (SCL signal to detect the component at 180º.)
            sda_module_180 : inout  STD_LOGIC;
                     (SDA signal to detect the component at 180º.)
            scl_module_270 : out  STD_LOGIC;
                     (SCL signal to detect the component at 270º.)
            sda_module_270 : inout  STD_LOGIC
                      (SDA signal to detect the component at 270º.)
                       );
end main;
```

B.11.a relays_map

This block uses the file "read_write_relay.vhd". This block controls the opening and closing of the external relays to create the desired routing system. This block uses the type of command decoded by the system.

The entity of the file is described as:

```
entity read_write_relay is
       Generic (
            N_relays : natural := 128
                 (Maximum number of relays per cell.)
                        );
    Port ( clk50M : in  STD_LOGIC;
                 (External clock oscillator @ 50MHz.)
           reset : in  STD_LOGIC;
                 (New internal reset in positive logic.)
           data_ready_PC : in  STD_LOGIC;
                 (Data ready from I2C from PC.)
           typeI : in  STD_LOGIC;
                 (A type I command is going to be processed.)
           typeIV : in  STD_LOGIC;
                 (A type IV command is going to be processed.)
           data_in : in  STD_LOGIC_VECTOR (7 downto 0);
                 (Data to be transfer.)
           data_out : out  STD_LOGIC_VECTOR (7 downto 0);
                 (Data read.)
           load_data_PC : out  STD_LOGIC;
                 (Load data from I2C from PC.)
           relays : out  STD_LOGIC_VECTOR (N_relays-1 downto 0));
                 (External relays to be controlled.)
end read_write_relay;
```

B.11.b send_IDs_block

This block uses the file "send_IDs.vhd". This block controls the information about the IDs of all the parts related with the cell: cell ID, neighbors (4 in total) IDs, modules, and also the orientation of the module.

The entity of the file is described as:

```
entity send_IDs is
       Generic (
            Nbits_data : natural := 8
                 (Number of bits for the data in the I2C connection.)
                        );
    Port ( reset : in  STD_LOGIC;
                 (New internal reset in positive logic.)
           clk50M : in  STD_LOGIC;
                 (External clock oscillator @ 50MHz.)
           typeII : in  STD_LOGIC;
                 (A type II command is going to be processed.)
           command_master : in  STD_LOGIC_VECTOR(7 downto 0);
                 (Decoded command sent by the PC.)
           my_ID, north_ID, east_ID, south_ID, west_ID, module_ID : in
STD_LOGIC_VECTOR(Nbits_data-1 downto 0);
                 (IDs from each cell, including the 4 neighbors and
                  the module.)
           module_orientation  : in  STD_LOGIC_VECTOR(Nbits_data-1 downto 0);
                 (Orientation of the module connected to the cell.)
           data_out : out  STD_LOGIC_VECTOR(Nbits_data-1 downto 0);
                  (Data read.)
           load_data_PC : out  STD_LOGIC);
                  (Load data from I2C from PC.)
end send_IDs;
```

B.11.c detect_map

This block uses the file "detect_type.vhd". This block decodes the command sent by the PC and generate the specific signals to control other blocks. It also activates specific flag signals depending on the type of command decoded.

The entity of the file is described as:

```
entity detect_type is
    Port ( reset : in  STD_LOGIC;
                (New internal reset in positive logic.)
           clk50M : in  STD_LOGIC;
                (External clock oscillator @ 50MHz.)
           busy_PC : in  STD_LOGIC;
                (I2C from PC is busy.)
           data_ready_PC : in  STD_LOGIC;
                (Data ready from I2C from PC.)
           data_in : in  STD_LOGIC_VECTOR (7 downto 0);
                (Data to be transfer.)
           TypeI : out  STD_LOGIC;
                (A type I command was decoded.)
           TypeII : out  STD_LOGIC;
                (A type II command was decoded.)
           TypeIII : out  STD_LOGIC;
                (A type III command was decoded.)
           TypeIV : out  STD_LOGIC;
                (A type IV command was decoded.)
           TypeV : out  STD_LOGIC
                (A type V command was decoded.)
          );
end detect_type;
```

This block uses a Xilinx IP block generated by the CORE Generator: Block Memory Generator. It's a Single Port ROM that stores all the commands and their types. The file used is "commands_mem.xco" and the memory initialization file is "commands_mem.mif".

B.11.d reading_module_map

This block uses the file "reading_module.vhd". It controls the reading and status of the module connected to the cell. This block controls 4 different master I^{2}C blocks to read the orientation of the module connected. It also makes a local copy of the datasheet of the module connected.

The entity of the file is described as:

```
entity reading_module is
    Port ( reset : in  STD_LOGIC;
                (New internal reset in positive logic.)
           clk50M : in  STD_LOGIC;
                (External clock oscillator @ 50MHz.)
           TypeV : in  STD_LOGIC;
                (A type V command was decoded.)
           data_in_PC : in  STD_LOGIC_VECTOR (7 downto 0);
                (Data to be transfer.)
           data_ready_PC : in  STD_LOGIC;
                (Data from the PC is ready to be read.)
           data_out : out  STD_LOGIC_VECTOR (7 downto 0);
                (Data read.)
           load_data_pc : out  STD_LOGIC;
                (Load data from I2C from PC.)
           orientation : out STD_LOGIC_VECTOR (7 downto 0);
                (Word to decode the orientation of the module connected.)
           scl_module_0 : out  STD_LOGIC;
                (SCL signal to detect the component at 0º.)
           sda_module_0 : inout  STD_LOGIC;
                (SDA signal to detect the component at 0º.)
           scl_module_90 : out  STD_LOGIC;
                (SCL signal to detect the component at 90º.)
           sda_module_90 : inout  STD_LOGIC;
                (SDA signal to detect the component at 90º.)
           scl_module_180 : out  STD_LOGIC;
                (SCL signal to detect the component at 180º.)
           sda_module_180 : inout  STD_LOGIC;
```

```
                (SDA signal to detect the component at 180º.)
           scl_module_270 : out  STD_LOGIC;
                (SCL signal to detect the component at 270º.)
           sda_module_270 : inout  STD_LOGIC
                (SDA signal to detect the component at 270º.)
                    );
end reading_module;
```

i) dual_memory

This block uses a Xilinx IP block generated by the CORE Generator: Block Memory Generator. It's a Dual Port RAM that stores the datasheet of the module connected.

The entity of the IP core is:

```
ENTITY dual_mem IS
      port (
            clka: IN std_logic;
                  (External clock oscillator to write @ 50MHz.)
            ena: IN std_logic;
                  (Enable for the RAM write.)
            wea: IN std_logic_VECTOR(0 downto 0);
                  (Write or Read.)
            addra: IN std_logic_VECTOR(9 downto 0);
                  (Address for the data to be written.)
            dina: IN std_logic_VECTOR(7 downto 0);
                  (Data to be written.)
            clkb: IN std_logic;
                  (External clock oscillator to read @ 50MHz.)
            enb: IN std_logic;
                  (Enable for the RAM read.)
            addrb: IN std_logic_VECTOR(9 downto 0);
                  (Address for the data to be read.)
            doutb: OUT std_logic_VECTOR(7 downto 0));
                  (Data to be read.)
END dual_mem;
```

ii) reading_0, reading_90, reading_180, reading_270

This VHDL file is a master block to control I^{2}C communications to read and detect the module connected to the cell.

The entity of the file " main_master.vhd" is described as:

```
entity main_master is
      generic (
            Nbits_data : natural := 8;
                  (Number of bits for the data in the I2C connection.)
            Nbits_address : natural := 7);
                  (Number of bits for the address in the I2C connection.)
   Port (  enable_master : in  STD_LOGIC;
                  (Enable signal.)
           readwrite_01 : in  STD_LOGIC;
                  (Signal to select if a Read or Write transfer
                   will be performed.)
           reset : in  STD_LOGIC;
                  (New internal reset in positive logic.)
           clk50M : in  STD_LOGIC;
                  (External clock oscillator @ 50MHz.)
           scl : out  STD_LOGIC := '1';
                  (SCL signal.)
           sda : inout  STD_LOGIC := '1';
                  (SDA signal.)
           load_address : in  STD_LOGIC;
                  (Internal signal to load the address to be sent.)
           address : in  STD_LOGIC_VECTOR (Nbits_address-1 downto 0);
                  (Address to be used.)
           load_data : in  STD_LOGIC;
                  (Internal signal to load the data to be sent.)
           data_in : in  STD_LOGIC_VECTOR (Nbits_data-1 downto 0);
                     (Data to be sent.)
           data_out : out
                STD_LOGIC_VECTOR (Nbits_data-1 downto 0) := (others => '0');
                     (Data read.)
           data_read : out  STD_LOGIC := '0';
                     (Data read ready in the register.)
           busy : out  STD_LOGIC := '1');
                     (Signal to indicate that the block is busy:
```

```
                                  a transfer is being processed.)
end main_master;
```

Appendix C: The AFRL-UNM HERC Prototype Details (from [1])

Here, we present the AFRL-UNM prototype in more detail, describe its hardware, its software, and discussion towards its miniaturization. Future users of the HERC will find this information useful to replicate it, enhance it, or to obtain a deeper understanding of the components utilized on it.

The AFRL-UNM HERC prototype is a system composed of two Basic Modules hosted by a data backplane, and a power backplane. This configuration, shown in Figure C-1, allows simulating and studying the characteristics of the larger HERCs.

Each Basic Module (BM) includes a Power Distribution System (PDS) integrated within its Printed Circuit Board (PCB). Modules also include components intended to facilitate the implementation and debugging of logic systems in its FPGA and to support the system's operation. With these components and integrated systems, a BM can be used as an independent dual processor reconfigurable system, or it can be easily integrated in more complex multiprocessor reconfigurable systems. We next describe the characteristics of the BM more in detail, and the techniques utilized in its design and manufacturing.

C.1 Description of the Basic Module Prototype

A photograph of the prototype AFRL-UNM HERC is shown in Figure C-2 with its main components labeled.

Due to the large number of components that the BM requires, the high component density, and the special characteristics of the high-speed traces in the PCB, the board was designed to utilize twenty layers, a surface area 8 inches by 6 inches, and a thickness of 0.114 inches of FR4-IS410 material. It is a through hole via and microvia board, with vias allowed in pads and plugged microvias under Ball Grid Array (BGA) components. This board has eight layers dedicated to supporting high-speed signal routing, two layers used for low-speed routing, nine solid planes for power distribution, and one layer used for miscellaneous power distribution.

Table C-1 identifies the layers of the PCB and their functionality, providing for both signal lines and power distribution. It is important to note that each layer used for routing high-speed signals is in between solid planes to aid in obtaining controlled impedance in the traces. Also, these high-speed lines, such as those used for the Rocket IO network, clock distribution and the DDR2 memory paths were carefully planned, measured, and routed. These signals had to be length matched, impedance controlled, and in some cases, terminated with specific resistances.

Table C-1. Layer and Signal Distribution on the AFRL-UNM BM PCB

Layer	Type	Traces
1	High-Speed Signals	Clocks, Other High-Speed
2	Solid Plane	Ground
3	High-Speed Signals	Rocket IO
4	Solid Plane	Ground
5	High-Speed Signals	Rocket IO
6	Solid Plane	Ground
7	Solid Plane	Power 1.2 V
8	High-Speed Signals	Memory A
9	Solid Plane	Power 2.5 V
10	High-Speed Signals	Memory A
11	Solid Plane	Power 1.8 V
12	High-Speed Signals	Memory B
13	Solid Plane	Power 3.3 V
14	High-Speed Signals	Memory B
15	Solid Plane	Power 0.9 V
16	Signal	Low Speed Signals
17	Power Traces	Mixed Power
18	Signal	Low Speed Signals
19	Solid Plane	Ground
20	High-Speed Signals	Clocks, Other High-Speed

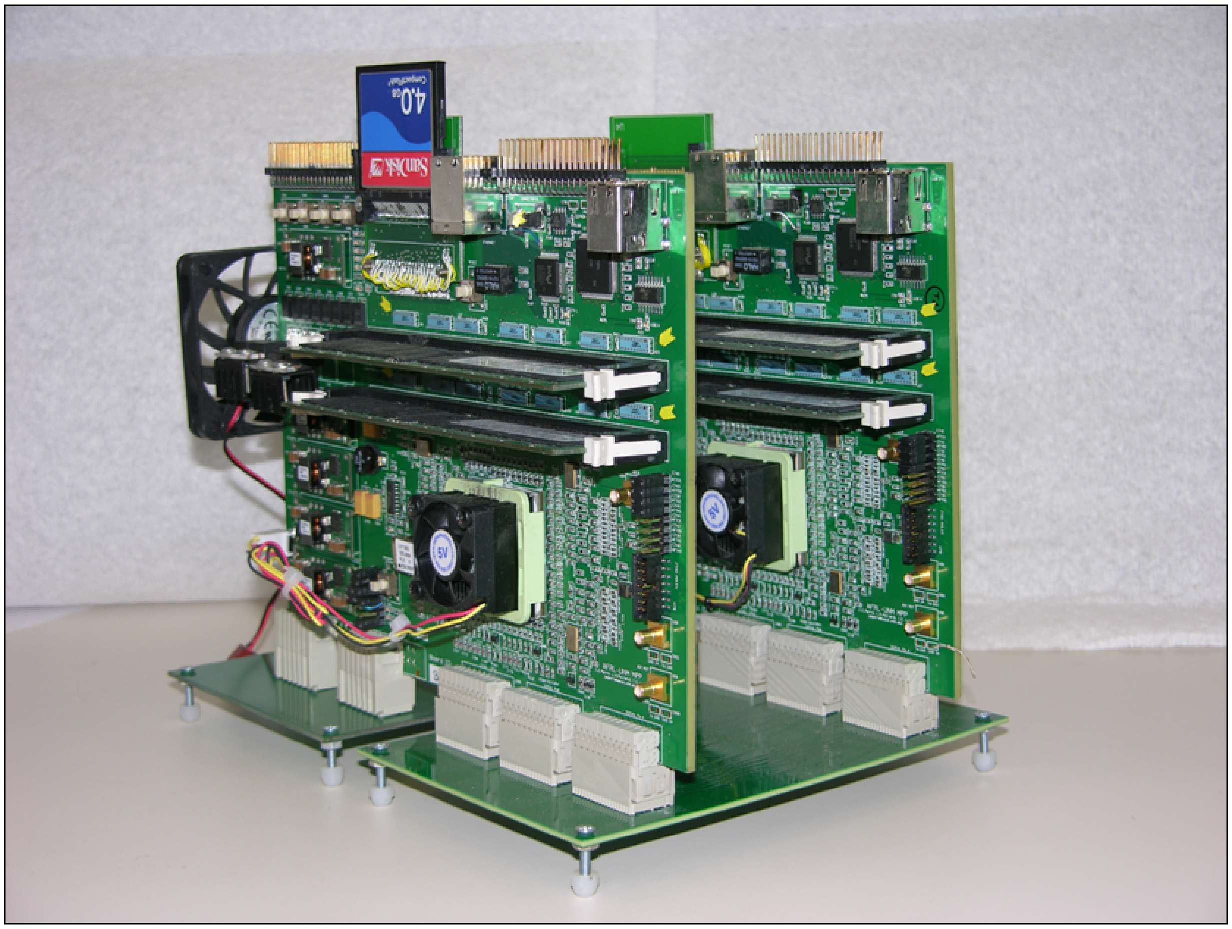


Figure C-1. The AFRL-UNM HERC Prototype.

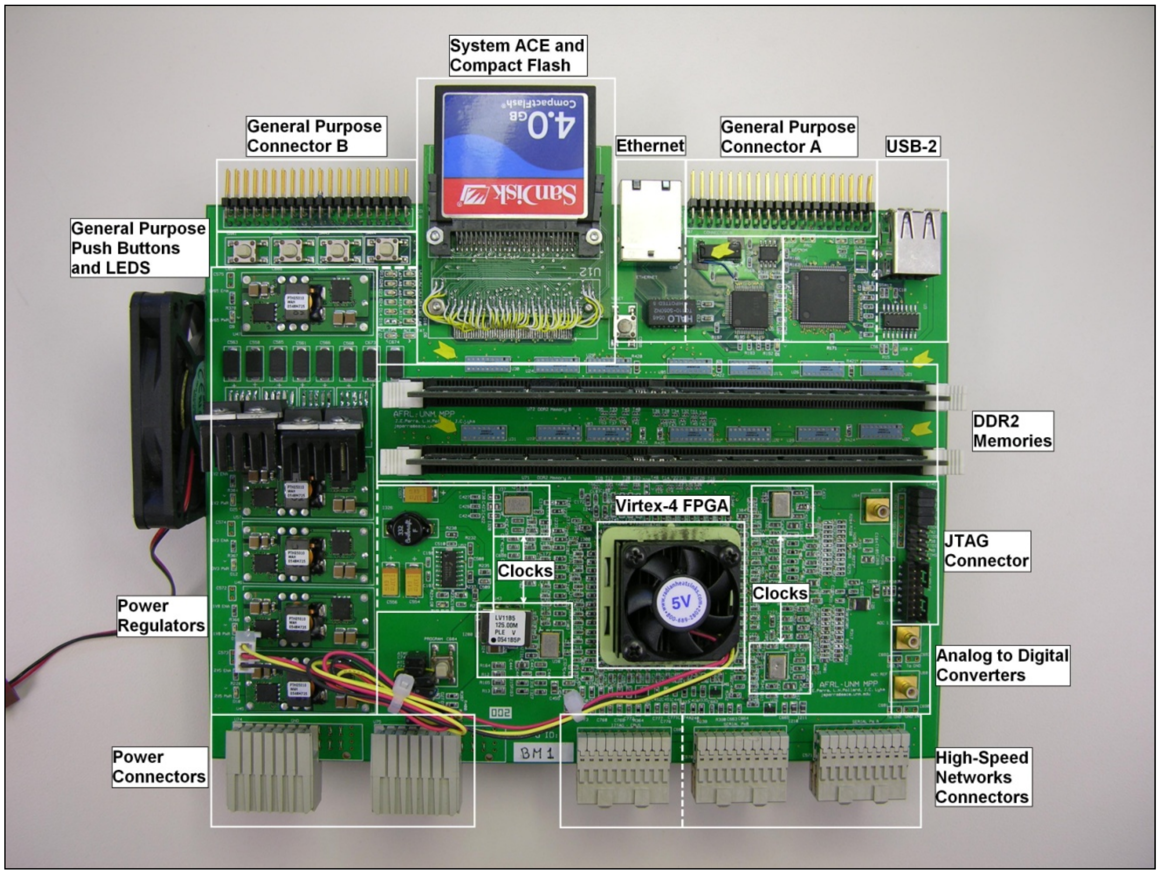

System ACE and Compact Flash
General Purpose Connector B
Ethernet
General Purpose Connector A
USB-2
General Purpose Push Buttons and LEDS
DDR2 Memories
Virtex-4 FPGA
Clocks
Clocks
JTAG Connector
Power Regulators
Analog to Digital Converters
Power Connectors
High-Speed Networks Connectors

Figure C-2. Main components of the AFRL-UNM BM.

C.1.1 Power distribution on the Basic Module

Designing and implementing the power distribution system in the Basic Module was not a trivial task. A first concern on the design of the PDS is that since the HERC is a reconfigurable system, the power consumption characteristics of the circuitry that will be implemented in the BM are not known. Therefore, the PDS had to be capable of providing low ripple voltages to a wide range of circuits with different power requirements. A conservative approach had to be utilized in its design, where a large number of bypass capacitors and wide power traces had to be used. Also, high-capacity voltage regulators had to be used to assure that large currents would be properly regulated.

A second challenge in the Power Distribution System design is to successfully generate the wide range of voltages the system requires, and to route them to the numerous and densely placed components in the PCB. High-speed communications modules, for instance, cannot be powered with the same switching power voltage adapters used for powering other components of the system. The use of linear regulators is necessary to satisfy their stringent power requirements. Therefore, the BM contains fifteen power regulators, five switching voltage regulators and ten linear voltage regulators.

The AFRL-UNM BM requires seven different voltages to be generated, as shown in the block diagram of the Voltage Generation System in Figure C-3. These voltages are generated from a high-current five volt input, and are distributed to the BM's components through solid planes, and through specially designed power traces buried in the internal layers of the PCB. Although the Power Distribution System was designed to provide power up to 100 Watts, the larger system tested in this manuscript utilized only half that amount. However, higher power consumptions are expected in circuits requiring a more extensive use of the FPGA resources.

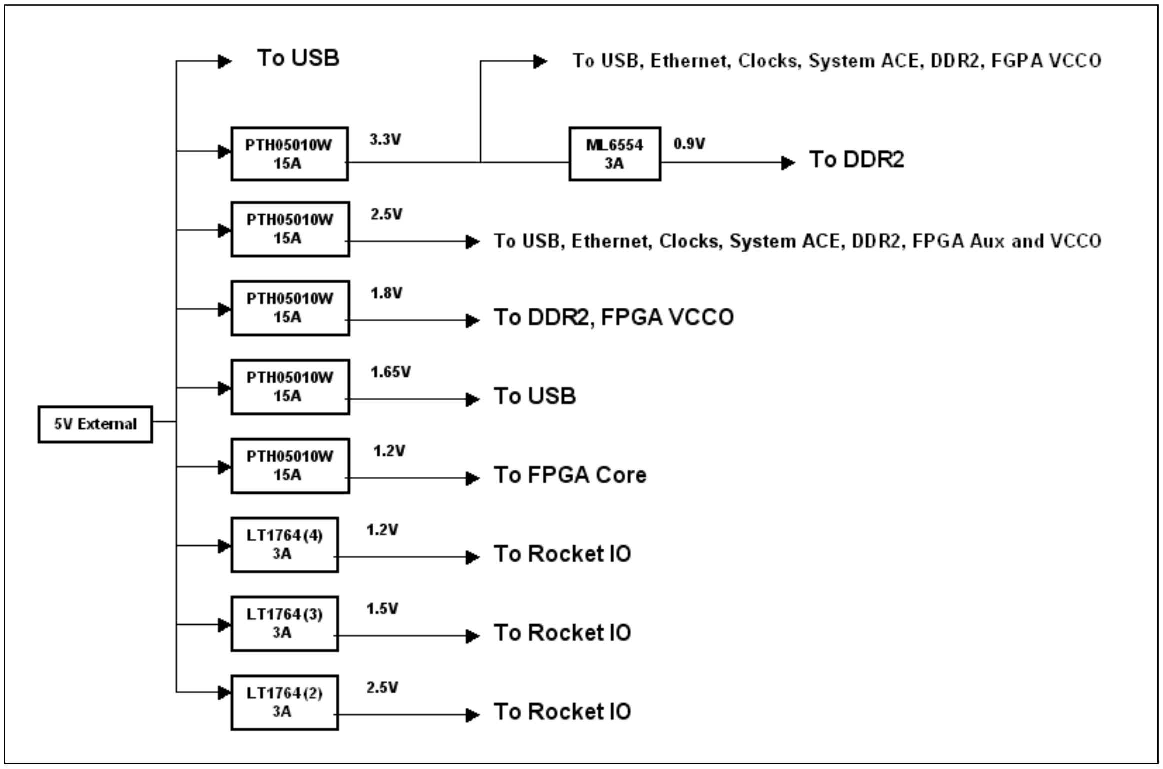


Figure C-3. Block Diagram of the AFRL-UNM BM Voltage Distribution System.

C.1.2 Clock Generation and Distribution System of the AFRL-UNM BM

The Clock Generation and Distribution System (CGDS) of the AFRL-UNM BM was designed to be flexible and sufficient for the general-purpose applications that could be synthesized in the HERC. It is composed of two synthesized external clocks, afixed external clock and four high-precision clocks used to feed the Rocket IO MGT modules. These clocks and the Digital Clock Management (DCM) units embedded in the FPGA allow multiple clock combinations to be utilized in the BM.

As shown in Figure C-4 and as mentioned above, two external clocks are synthesized. In this case, they are externally generated from a 33 MHz clock by a clock synthesizer, and linked to a global clock input of the BM's FPGA. This generated frequency can be chosen through a parallel or a serial input of the clock synthesizer. In the AFRL-UNM BM, the parallel input can be externally set using resistors, while the serial input is driven by the FPGA. This clock strategy, commonly used in high-speed systems, allows a dynamic change of the externally generated clocks. Despite the parallel ports of the clock synthesizers were fully implemented in the BM, communications with the serial port are left for future implementations.

C.1.3 JTAG Chain

The JTAG chain of the HERC was designed to allow multiple options for device interconnection. In a BM, it can be configured to chain a JTAG port to the System ACE and to the FPGA, or to only interconnect the JTAG port and the FPGA. The JTAG chain can also be set to allow multiple BM devices to be interconnected. For example, a chain composed of the JTAG port, the System ACE and the FPGA in a BM, and a second FPGA in the other BM is possible. Using this option, multiple FPGAs can be configured from a single Compact Flash or JTAG port, and can be debugged from a single port.

C.1.4 The DDR2 Memories and Memory controllers

The BM features two PC2-4200-R DDR2 memories independently attached to the Virtex-4 FPGA. These memories can be paired with each embedded PowerPC, can be used by one of these microprocessors, or can be used by customized systems implemented in the FPGA.

The DDR2 memory layout was implemented to conform to the specifications of the most demanding memory controllers. In a DDR2 memory, the signals that are utilized in transferring information are divided into four groups, each with different routing and spacing requirements.

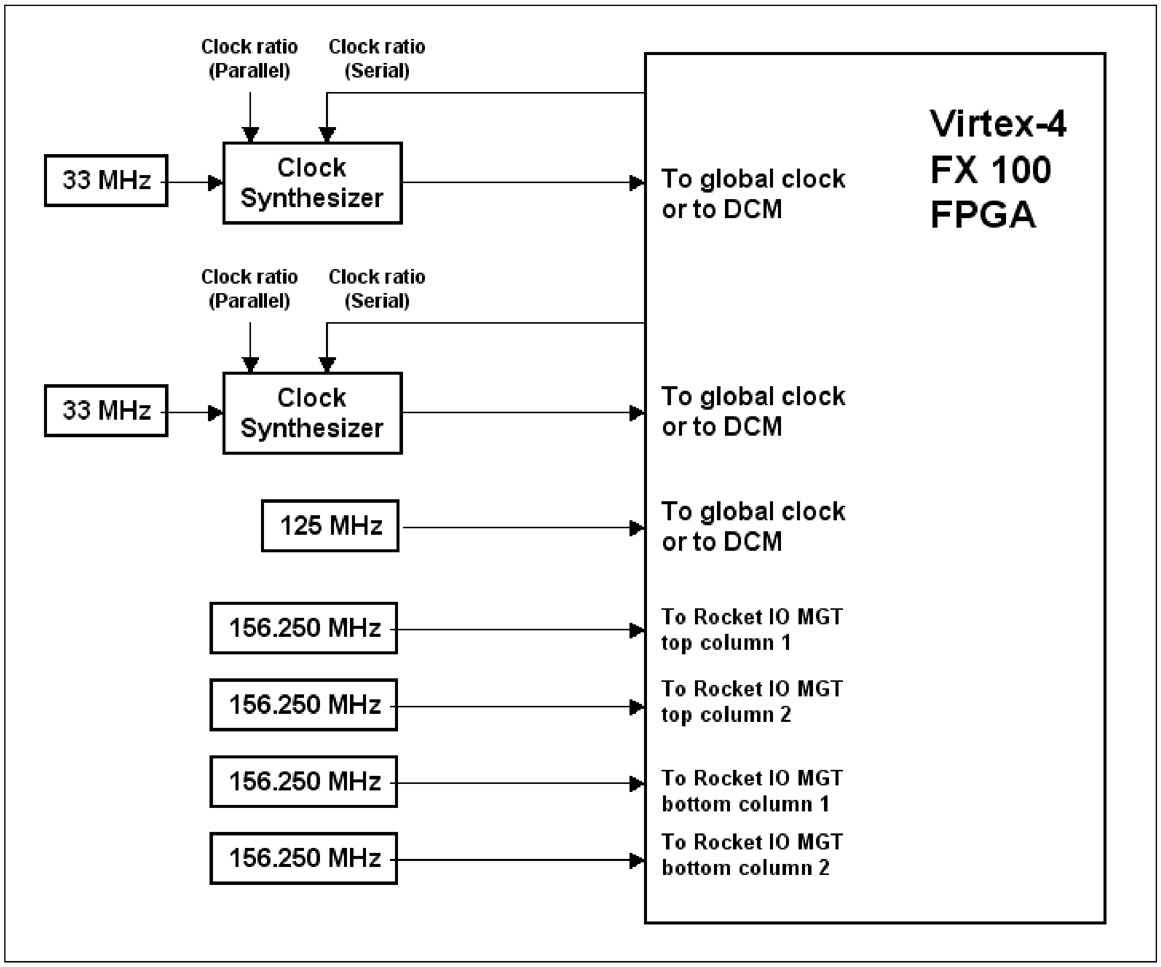


Figure C-4. Clock Generation and Distribution System (CGDS) of the AFRL-UNM BM

These are a Data Group, an Address and Command Group, a Control Group, and a Clock Group. In the AFRL-UNM BM, each group was routed separately from the lines that composed other groups, and all the signals in each group were length matched.

Length matching the signals in the data group required a considerable effort, since this group is composed of over a hundred lines. Concerned about this requirement, the large amount of space utilized, and its design and manufacturing cost, memory designers have reorganized the data lines into smaller groups for newer memory controllers. These group lengths need only to be matched among the lines that compose a group, and to an extra line added as a signal strobe. Unfortunately, the routing of the memories in the AFRL-UNM BM did not allow using that advantage.

To interface the DDR2 memory and any other circuit in the FGPA, it is necessary to use a memory controller. This is a hardware module that provides the necessary commands to initialize, read and write the memory and handle memory refresh cycles. Currently, Xilinx provides a memory interface generator used when the memory controller will be attached to a microprocessor, and another memory interface generator used for general-purpose cases. They are the PLB Double Data Rate (DDR2) Synchronous DRAM (SDRAM) Controller, and the Memory Interface Generator (MIG) respectively. At the time the BM board was designed, MIG required all signals in the DDR2 memory bus to be length matched, and half-length matched to an extra looped line. Therefore, to allow the use of all possible memory controller generators, the routing of the AFRL-UNM BM complied with this requirement. Figure C-5 shows a detail of one of the layers used for the DDR2 routing, where the meandering routes used to length match the traces and the aforementioned looped line are shown.

Using the PLB Double Data Rate (DDR2) Synchronous DRAM (SDRAM) Controller in the HERC required a collaborative effort with Xilinx to debug it and correct some of its characteristics. Once a working system was obtained, ten different tests were applied to exercise the DDR2 memory. These tests are a ChessBoard test, its reversed version, a 0s&1s test, an incremental data value test, the address reversed versions of these tests, a four memory positions-ahead test and a Xilinx-provided memory test.

In a ChessBoard test, even memory positions are written with an alternate series of one and zeroes (starting with a zero). Odd memory positions are written with similar data, but starting with a one. Then the memory is fully read, hoping that no discrepancies between the written and the read information are found. The reversed version of this test was also applied. In this case, the same testing procedure is performed but reversing the values written into odd and even memory positions. Figure C-6.a shows that the patterns formed in the memory by this test resemble a chessboard.

0s&1s memory test is a simple test where every odd memory positions are filled with 0xFFFFFFFF. In a similar fashion, every even memory position is filled with zeroes. As in the previous test, once the memory is written, it is read, and their values are compared looking for consistent memory records. The reversed version of this test is similar in procedure. Figure C-6.b shows the memory patterns formed in memory by this test.

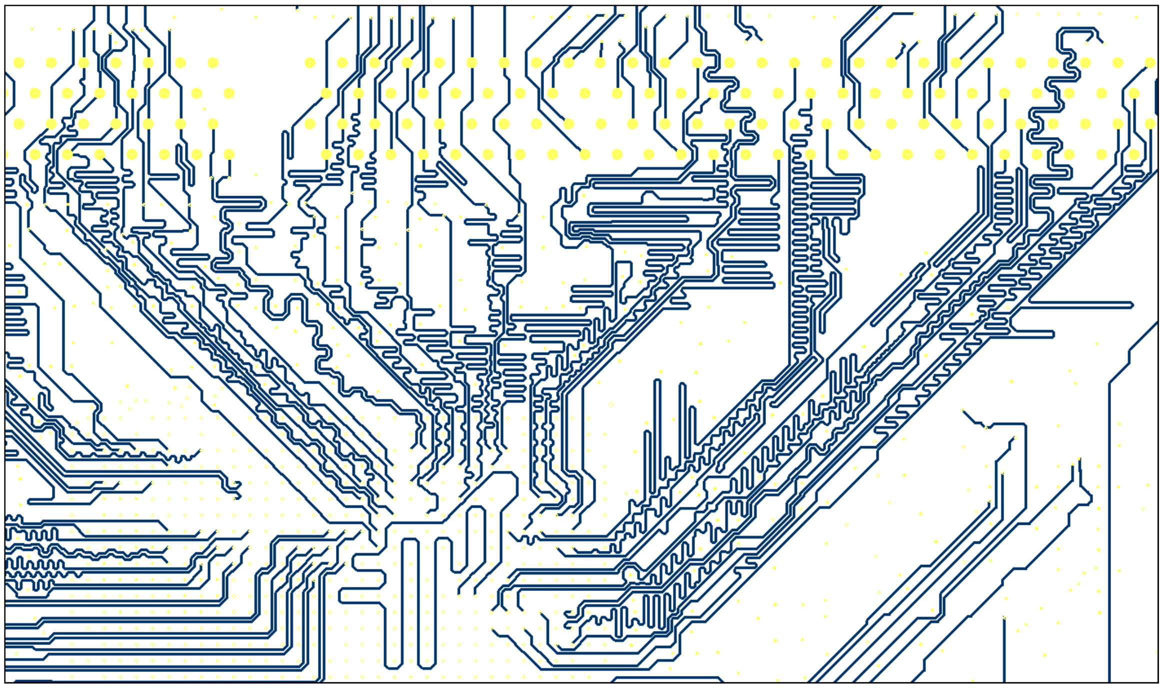


Figure C-5. Detail of the routing of the AFRL-UNM BM DDR2 memory bus

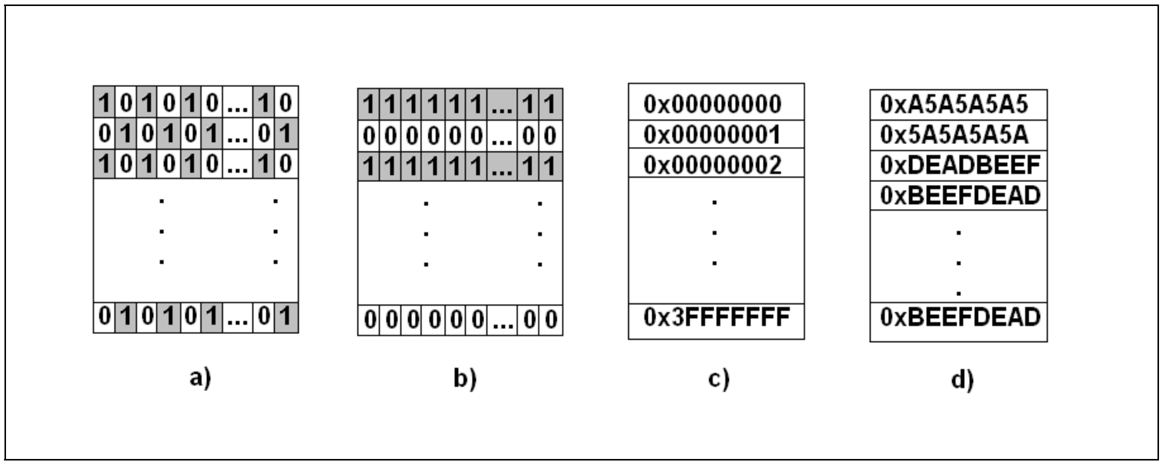


Figure C-6. Data Patterns obtained in different memory tests. a) ChessBoard Memory test, b) 0s&1s Memory test, c) Incremental Data Memory test, d) Four Memory Positions-Ahead test

The incremental data value test saves different numbers in different memory positions. Commonly, it is used to tag each memory position with the value of its address. Figure C-6.c shows an example of the data recorded with this test. Its address-reversed version consists on repeat it starting from the last position of the memory and ending in the first position.

The last two memory tests applied were a four memory positions-ahead test and a memory test provided by Xilinx. The first one consists on sequentially writing four different values on four consecutive memory positions. This is illustrated in Figure C-6.d. The test continues by reading these memory locations and assuring data consistency. It is intended in this case, to evaluate the memory's capability for writing and reading bursts of data. As interleaved, burst-oriented memories continue to be a standard, verifying this feature is becoming important. Finally, Xilinx provided memory test reads and writes a sequence of four random values in bytes, double-bytes, half and full word to four random memory locations. These memory positions are read and compared with the written values looking for data consistency.

Besides verification of successful data storing and retrieval, the correct behavior of the memory system was validated for a small range of clock frequencies and distributions. The memory controller has been tested for bus clock frequencies of 100 MHz on the PLB version and 100MHz and 66MHz in OPB version, and for DDR2 clocks of 133, 200 and 266 MHz. Given the dual clock data rate featured on DDR2 memories, 266, 400 and 533 MHz can be achieved for data transfer. This last frequency is the maximum one used by the BM DDR2 memory system. Clock distributions of 200 and 266 MHz with a 100MHz PLB clock frequency were also successfully verified.

C.1.5 High-Speed networks

The prototype of the AFRL-UNM BM has three high-speed connectors; two used for interconnecting twenty Rocket IO communication links to other BM boards, and one carrying the links for a common bus network and the JTAG chain. Although these connectors are designed to carry data transmissions up to 6 Gbps, successful performance up to 10 Gbps has been reported. In the AFRL-UNM HERC, successful data transmission using the Rocket IO links has been measured at a rate of 6.25 Gbps.

The traces used for Rocket IO, as well as half of the common bus network, were designed to support high-speed data rates. Traces with 50 Ohms controlled impedance and 100 Ohms cross-impedance were used to interconnect the Rocket IO to its connectors and other transceivers. In the design process, special care was taken to utilize an intact reference plane in order to avoid impedance discontinuities. No vias, except at the beginning and end of the Rocket IO, lines were used. Also no layer-to-layer changes were allowed for the high-speed signals. Every Rocket IO pair was separated from other lines to diminish electromagnetic interference between them. These characteristics allow in theory reaching speeds of 10.3125 Gb/s through 16 inches of FR4 material. However the material used in the AFRL-UNM BM (FR4-IS410), although still an FR4, has improved dielectric characteristics over other materials in its family, and it is expected to be able to handle higher speeds in longer traces.

C.1.6 System ACE

The AFRL-UNM BM includes a System Advanced Configuration Environment (System ACE) that supports Compact Flash Memories with data storage capabilities up to 4 GB. The System ACE is a module composed of a controller device (ACE controller) and a Compact Flash storage device (ACE Flash). It provides a configuration solution for one or more FPGAs, and in addition, a microprocessor interface for utilizing the ACE Flash as a peripheral.

In the AFRL-UNM BM, the System ACE can be used to configure, upon reset, every Virtex-4 FX 100 in the JTAG chain FPGA. The System ACE is therefore, the first device that should be encountered in the JTAG chain of a series of BMs. Once configured, each BM can use the system ACE on its board as storage peripheral. For this purpose, the ACE Flash is divided into three partitions, two used as a filesystem, and the remaining one used for boot configuration.

C.1.7 Ethernet support

Each AFRL-UNM BM contains an Intel LXT971 PHY device operating at 10/100 Mb/s. This device is connected to the FPGA, and to its embedded Ethernet MAC, through a MII interface. The PHY is connected to a HALO Magnetic Isolation Module and a RJ-45 connector. To implement a small Ethernet network test platform, each BM was connected to an Ethernet switch with routing capabilities. This network was complemented with two dedicated Personal Computers (PC). These PCs were then used to remotely access and exercise the BMs.

To verify the correct functionality of the Ethernet module, Xilkernel, a light, tunable OS kernel that supports the POSIX standard and processes scheduling strategies, was used to implement a small Webserver on the AFRL-UNM BM. Xilmfs and Lwip were executed by this OS kernel, providing the application support for file systems and TCP/IP stacks. With this webserver, TCP requests were listened and answered through the HTTP port, allowing general-purpose pins to be remotely sensed and controlled through a web page.

C.1.8 Other Peripherals and connectors on the BM

Other peripherals on the AFRL-UNM BM are a host USB-2 Serial Communications port, two10-bit / 200 KSPS Analog to Digital Converters (ADC), a group of general-purpose LEDs, and Switches. Each BM has also two general-purpose connectors, used to host external expansion boards. Using one of these connectors, two RS-232 UARTs and an extra II-C flash memory have been added to the Basic Module.

Although fully implemented in the BM, the testing and the integration of the USB-2 port and the ADCs to the OS are left for future work. These peripherals were included to allow future users of the HERC to add more capabilities to the system, such as external Hard Disk Drives (HDD), wireless interconnection or remote sensor data capturing. The Basic Module can also be expanded with eight more ADCs, using some lines shared with one of the DDR2 memories. Therefore, a PCB containing these ADC connectors should have the same dimensions as a DDR2 DIMM and should be used in place of the memory DIMMs.

C.2 Backplanes and Physical Scaling of the HERC

As mentioned above, the HERC prototype includes two backplanes, one for power distribution, and one used to carry data signals. The power distribution backplane is manufactured with 4 oz, dual side cooper planes intended to not only provide a high current transport (20 A) but also to create a large capacitive effect. They utilize four female 90 degrees connectors, mated to the power socket connectors of the BM, and two input power connectors. The data transport backplane features 20 differential pairs with 50 Ohm controlled impedance lines, and 100 Ohm cross-impedance lines used for Rocket IO serial communications. They cross-interconnect every high-speed output on each BM's high-speed connectors. The data transport backplane also features 16 single low-speed lines for data broadcasting, and 4 traces used for JTAG board interconnection.

Using the same interconnection strategy, these backplanes can be easily expanded to host more BM boards. Figure C-7 shows a conceptual model of a system composed of five Basic Modules in a single backplane. It is left for future research to investigate the feasibility of scaling the HERC to systems with larger numbers of BMs. However, Figure C-8 and Figure C-9 demostrate how these systems could be arranged in situations where space is a major constraint.

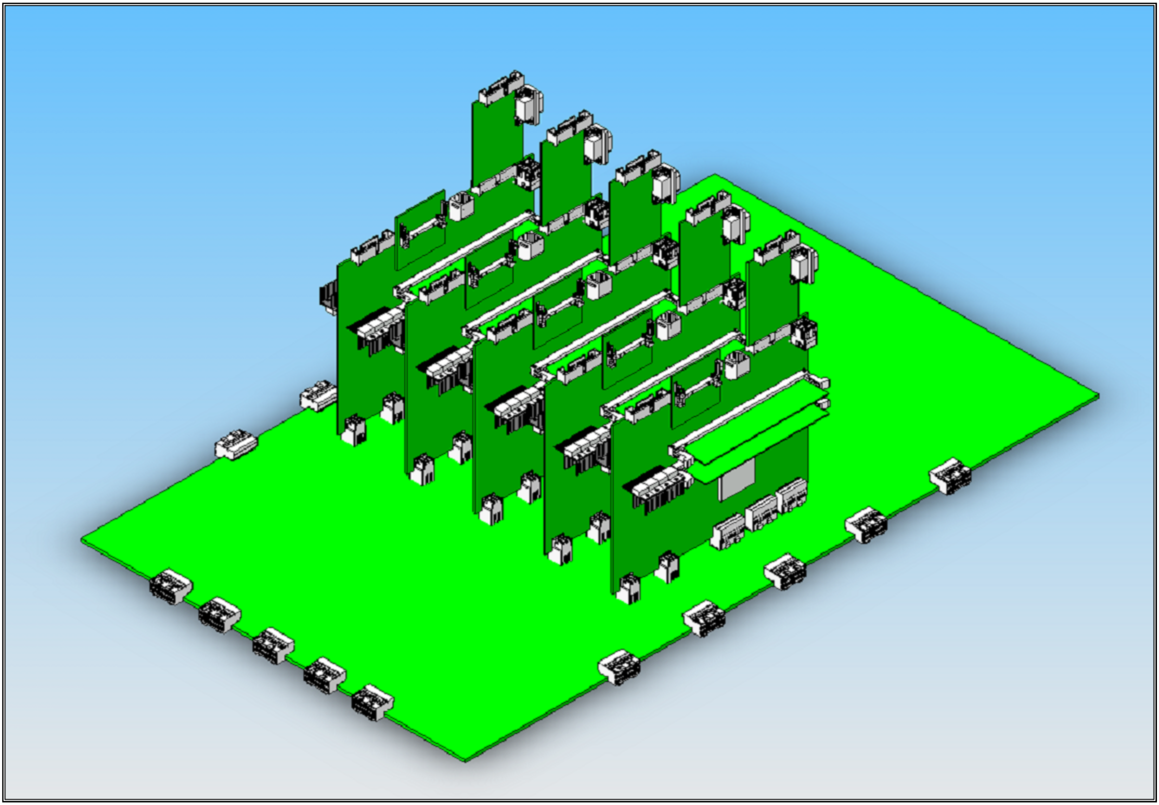


Figure C-7. Conceptual model of a HERC composed of five BMs

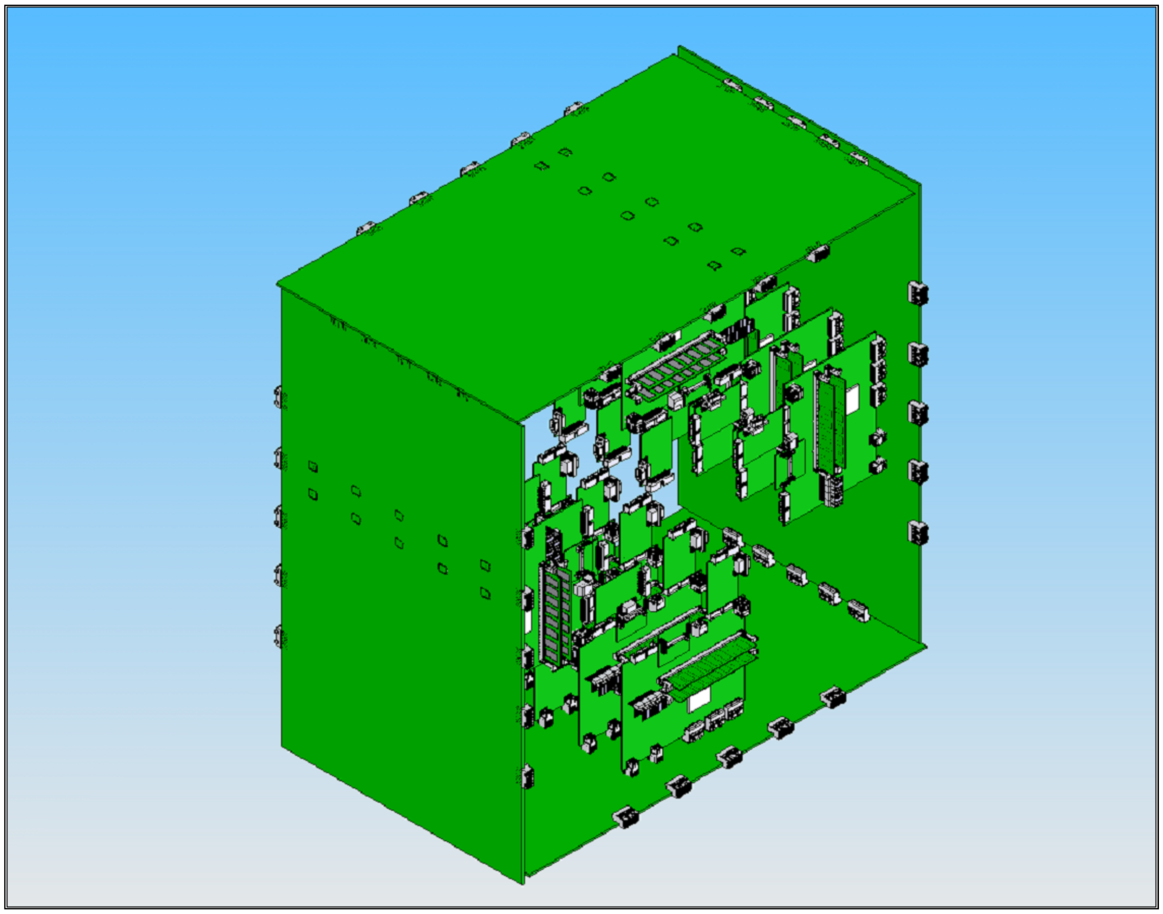


Figure C-8. Conceptual model of a HERC with twenty BMs

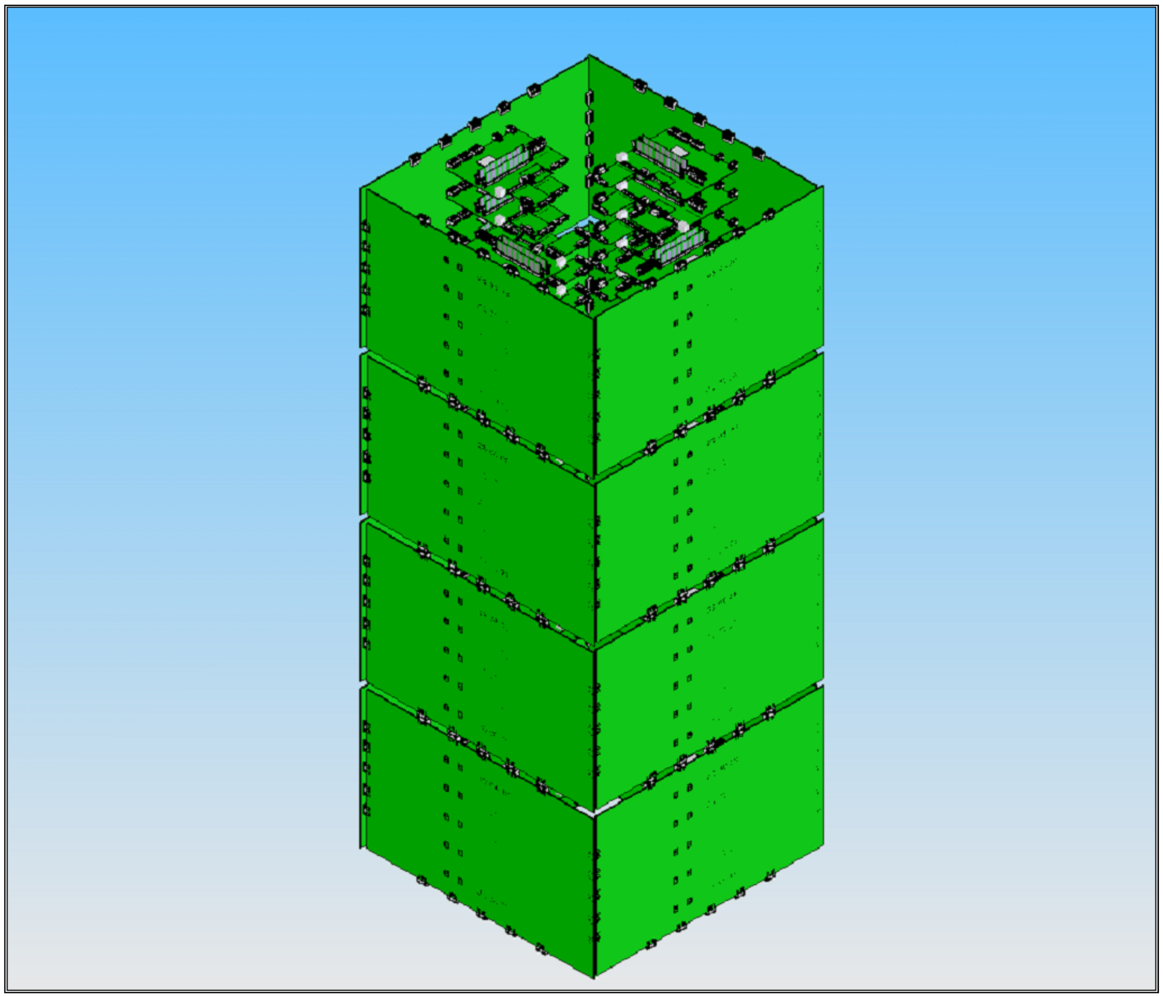


Figure C-9. Conceptual model of eighty BMs in a HERC

C.3 Description of the HERC's software

The software portion of the HERC is mainly composed of Montavista Linux as its Operating System, OpenMPI running over RSH used as the message passing engine, and a networking driver, which is a Linux kernel module used to interface the high-speed networks with the Operating System.

C.3.1 Linux on the AFRL-UNM HERC

Montavista Linux 4.0.1 with kernel 2.6.10, and Montavista Linux 3.0.1 with kernel 2.4.20 were ported to the AFRL-UNM HERC. A customized Linux kernel version was also compiled and ported to the system from the kernel sources available at kernel.org. However, the development of this distribution was not continued due to the excessive amount of effort necessary to port every software packet required by a fully functional OS. Although Montavista Linux can be successfully used in future versions of the HERC, its high cost resulted in continuing development utilizing another Linux distribution.

C.3.2 The Networking Drivers

The Networking Drivers developed for the communication links of the AFRL-UNM HERC are based on the Network Utility that is a functional template that provides examples on how to register a driver against the kernel, how to initialize it, open it and clean the kernel when stopping it.

The first activity executed when the network driver is inserted in the operating system is to create and register a networking device. A name is assigned to that device, and its communications structure is created. A second step on the successful configuration of this driver involves the user's action. The user assigns some properties to this driver, such as its IP address, network mask and its status on the system. Using this information, an application wanting to use the communications hardware associated with the networking driver simply identifies the IP address of receiver party. The Operating System calls the driver passing the information to be transmitted to the driver's communications structure.

The driver's initialization also includes setting up the communications hardware. It reserves I/O addresses, registers the communication peripheral's interruptions to the operating system, and sets the involved peripherals to be ready for transmission and reception of data. Finally, the driver also contains the interrupt service routine (ISR), which is involved when data arrives at the communications hardware.

The networking driver and the hardware implemented for the high-speed communications were designed to reduce the amount of computation time dedicated by the microprocessor in the communication process. In low-speed communication devices, an interrupt request is issued and processed each time a character arrives. However, if this strategy would be used in high-speed, congested networks, the processor would be constantly interrupted, consuming most of their processing time. To alleviate this situation, a strategy combining register pooling and interruptions is used. In this case, FIFO structures are used in the hardware reception paths, only interrupting the processor when the FIFO is partially full. The driver then reads all available characters in the FIFO.

The communications hardware of the AFRL-UNM HERC implements a packet control module that prevents the microprocessor from being interrupted until the reception of an Ethernet packet is identified. In a common situation, the networking driver reads a group of characters from the receive FIFO, and provisionally stores them in memory. It only sends the characters to the OS when a complete Ethernet packet is received. The implemented system takes advantage of that by mapping this process into hardware, executing the aforementioned process without the microprocessor's intervention.

C.4 Miniaturization of the AFRL-UNM HERC

Initial efforts were started to miniaturize the AFRL-UNM HERC. General Electrics Research (GE Research), with the collaboration of the Embedded Systems and DSP laboratory of the Electrical and Computer Engineering of the University of New Mexico, started the miniaturization of the DDR2 memories of the HERC. Here, a three-dimensional stack composed of the DIMM components was to be created. An initial prototype were to be developed in which a simplified version of the stack would be mounted in a conventional DDR2 DIMM board and hosted by one of the memory sockets in the AFRL-UNM BM. This prototype would prove the feasibility of this solution, while allowing the researchers to study the challenges this solution presents.

Simultaneously, a first approximation to a denser packaging for the AFRL-UNM HERC was proposed. In this version, shown in Figure C-10, the main components and modules of the system will be enclosed in a protective shell, surrounded by connections to the HERC networks and to its external peripherals. Although this approximation still requires studies on the integration of complex systems such as the HERC and on its power dissipation, the prototype it proposes could led to significant advances on the implementation of HERCs with large numbers of BMs.

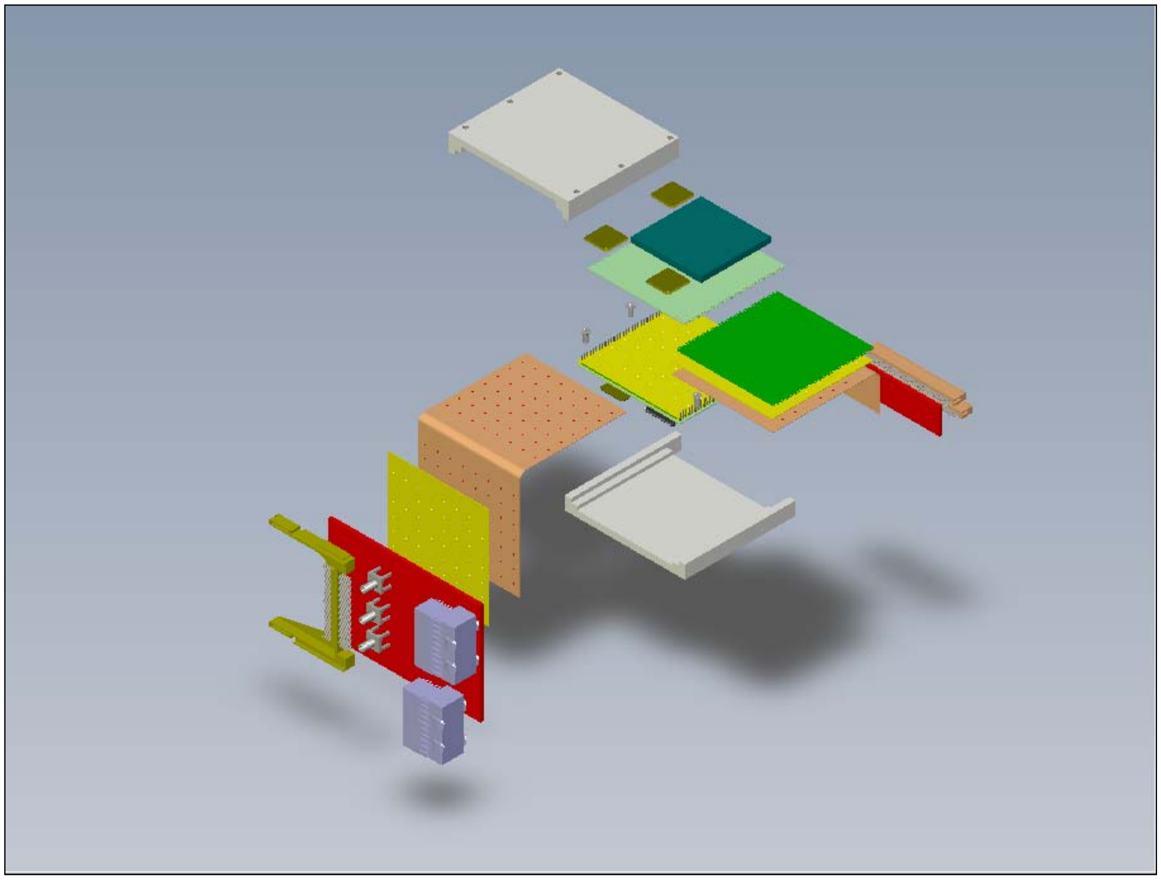


Figure C-10. Model of the proposal of an advanced packaging of the AFRL-UNM HERC BM (Used with permission of General Electric Research).

DISTRIBUTION LIST

DTIC/OCP 8725 John J. Kingman Rd, Suite 0944 Ft Belvoir, VA 22060-6218	1 cy
AFRL/RVIL Kirtland AFB, NM 87117-5776	2 cys
Official Record Copy AFRL/RVSE/Keith Avery	1 cy